The Truth About Disc Files

Eugene Volokh
VESOFT Consultants
Los Angeles, California

I/O, I/O, it's off to disc we go . . .
(modern rendition of Walt Disney)

ABSTRACT

The disc file is probably the most important part of MPE; however, due to the large number of different options and considerations inherent in disc files, these objects are often "under-understood" — this paper will try to present the truth and nothing but the truth (the whole truth will not be printed owing to lack of paper) about disc files, which will hopefully remedy this situation.

CHAPTER I FILE STRUCTURE

Where It's At

Before discussing disc files themselves, we must take a moment to point out some terms, probably already known to you, regarding the physical medium on which disc files reside — the disc. This disc consists of a lot of 128-word SECTORS, and is assumed to be configured on the system as one logical device.

Some considerations to be judged when referring to these discs are: (1) space — each disc has an ever so finite amount of sectors on it, the number of which varies from disc to disc, but is, by Murphy's Law, never enough — and (2) speed of access, which is typically on the order of 30 disc accesses per second.

The discs typically used with the HP are ones that constantly rotate in order for all parts of the disc to be accessible by the unrotating DISC HEAD. It is this rotation of the disc that is the culprit in the slowness of disc accesses. Similar considerations can be applied to the two other significant types of hardware: memory (which is very, very fast yet lamentably limited — up to 4 MegaBYTES on a Series 44) and tapes (which are virtually infinite yet quite slow).

The above hardware considerations, though elementary, will be of paramount importance in further discussion.

The Extent Question

Let us start at the beginning — the creation of the file. We will examine what the MPE operating system has to do to create a file. For example, let us say that you ask MPE to build you a data file, which is to have room for at most 100,000 records of 128 words each (note that 128 words is the size of a sector, and thus a good value for simplicity). This would be done, perhaps, by an MPE command akin to ":BUILD ING;DISC=100000" (MPE will automatically assume 128 words as the record size). Now, what does MPE do?

Well, of course, MPE must allocate some disc space for that file. In this particular case, MPE must allocate a whopping 100,001 sectors (the 1 extra sector is for the file label, a place where MPE holds internal file information like the lockword, etc.) all at one time. But, wait a minute! There may be 100,001 sectors out there on your disc (or discs), but it's possible that there is no one single gap that large out there. Moreover, maybe you don't really need all that space. Quite probably, you'll never use more than 10% of it! So, we are faced with a dilemma — if MPE were to allocate the space for that file nicely and simply, in one big chunk, it may not have enough space on disc; or, if it does, most of that space will probably be wasted, as (for a time, at least) you will not use all of that space.

Let us look at the other "extreme" solution. Why don't we, perhaps, allocate only one sector of space at a time — one in the beginning, for the file label, and one every time the user needs one. That way, even if the disc is hopelessly fragmented (i.e., there are very many 1-sector pieces of free space out there, but no large ones), we can probably fit a sector — if we can't, time to buy another disc; moreover, we do not allocate any disc space until we really need it. This was, perhaps, a decent solution in the "good old days" when disc space was very expensive. But, now, the operating system would have to maintain 100,001 pointers to enable access to that file, which makes the above method unworkable.

Enter the EXTENT! The extent is a reasonable compromise between the two extreme methods outlined above. A file can consist of anywhere from 1 to 32 extents (the default number is 8). Now, when we build the above file (with 8 extents), we will only have to allocate around 12,500 sectors in the beginning (a savings of disc space) and allocate new extents only every 12,500 records (a savings of disc accesses). We could, however, allocate the file with only 1 extent, thus losing out on disc space but gaining on disc accesses (but, of course, the savings on disc accesses is rather small compared to the incredible wastage of disc space), or with 32 extents, thus saving disc space at the expense of a few extra disc accesses.

Two other considerations come into play, however — one is that accessing files with a lot of extents FRAGMENTS THE DISC (i.e., increases the number of small holes at the expense of large holes), thus making new files harder to allocate in the future, and another is that it is better to run out of disc space when building a disc file, than when allocating a new extent in the middle of the program (precious time and internal data consistency may be lost this way). The former can be handled best by decreasing the number of extents (at the expense of, of course, disc space) and the latter by allocating at :BUILD-time all of the specified extents (but only if you are sure you will use all of the space). Note that the number of extents (maximum and initially allocated may be specified on the :BUILD command's DEV keyword, whose format is "DEV=device[,maxexts][,initalloc]", where maxexts defaults to 8 and initalloc to 1.

For Those With Multiple Discs

If you are the proud owner of several disc drives, another factor comes into play. For example, let us say that you build a file with the command ":BUILD ING;DISC=100000" (note that the maximum number of extents defaults to 8, 1 initially allocated), and start to wonder about which disc your file resides on. Well, MPE, has adopted the so-called "eeny, meeny, miney, moe" algorithm. That is, if you succeed in filling all 8 extents of your file, you may well find that that file does not reside on just one disc; rather, it resides on the discs of the DEVICE CLASS "DISC" (which are special sets of different devices, not necessarily discs, configured at system set-up time). Each extent, of course, resides wholly on one disc; but, the extents may reside on different discs — thus, a file with 8 extents may well find itself with 4 extents on disc #1, 2 on disc #2, 1 on disc #3, 1 on disc #4, and 0 on disc #5. If you, however, want that file to reside exclusively on disc #4, "no sweat" (as is said in the vernacular)! Merely :BUILD the file with the "DEV=4" parameter. Or, if you set up another device class called PRODDISC which will contain discs #3, #4, #5, building the file on DEV=PRODDISC will ensure that all extents of that file will be located on one of those devices. What, you may ask, is the importance of this? Well, the word that has leaked down from HP is: SPREAD OUT YOUR FILES — for instance, if you have two heavily accessed files, it might be wise to put them on two different discs.

This is done for the following reason. Let us assume that you have two disc drives, each one able to perform approximately 30 I/Os per second, and you spread out your files in such a way that each disc gets about 30 I/O requests per second. Those requests will be executed within one second. But, if there are 20 I/O requests per second to one disc and 40 to the second disc, the first disc will not perform up to capacity, and 10 of the requests to the second disc will have to wait for a second or more, thus degrading system performance.

Another promising idea is to configure all of your devices except the system disc as device class "DISC," thus keeping files off the system disc, and thus reducing the amount of access to the system disc, which already has the operating system and the virtual memory on it. However, with MPE IV, in which you will be allowed to spread virtual memory over several devices, this may not be as important. Note that for easy file disc location handling, MPEX/3000's %LISTF ,4 and %ALTFILE commands and ADAGER's DBCREATE and SETMOVE functions should be used.

The Logical File Structure

Besides the physical file structure described above — extents, sectors, etc. — MPE files also have an internal logical structure, not enforced in most ways by the actual file contents but rather by certain logical file descriptors like the record size, the blocking factor, the block size, the file type, and the like. First of all, we will discuss the simplest sort of MPE file — the fixed-record length file.

The Fixed Record Length File

A file is more than just a collection of data placed out on disc. It usually has certain logical relationships within it. One of the most frequent and fundamental relationships is one in which data is organized into chunks (called RECORDS) of a fixed length; for instance, if you have a data file which contains, for each customer, the customer code (6 characters), customer name (30 characters), and the amount owed you by the customer (8 zoned decimal characters), you have a 44-character entry for each customer. Therefore, it would be logical, for the sake of ease of access, to build that file with 44-byte (or 22-word) records, having one record per customer. So, to build that file, you would perform a BUILD command with the REC=−44,,F,ASCII parameter (− stands for bytes and F for fixed record length).

The Block

A familiar example of fixed record length disc file is your usual EDITOR /KEEP-NUMBERED file, a file with a record size of 80 bytes = 40 words. However, do you know that in your EDITOR keep files more than 6% of all disc space they occupy is wasted? This may not sound like much, but if you are running short on disc space, this can be a lot. What's more, that disc space can be saved (for large files) by merely specifying a certain :FILE equation for the file to be kept. What, you may ask, is the reason for this wastage? Well, the answer lies in the secrets of the BLOCK.

The fundamental unit of disc I/O (as far as MPE is concerned) is the SECTOR (128 words). Practically all disc I/O ends up as multiples of 128 words. 40, of course, is not a multiple of 128. So, if MPE decided to place 40 words per sector, it would waste not 6%, but 69% of each sector! So, you ask, why not pack three 40-word records into one 128-word sector. Well, that's exactly what MPE does; but because 128 is not a multiple of 40, either, it still wastes 6% of the file's disc space (although 6% may not sound like much, for some unlucky files which have different record lengths, it can be worse, with up to 50% wasted space!). But, there is light at the end of the tunnel! We can very snugly fit 16 40-word records into 5 128-word sectors — a perfect fit.

From the above labyrinth come the notions of the BLOCKING FACTOR and the BLOCK. The BLOCKING FACTOR is, very simply, the number of records that we choose to fit into a multiple of 128 words — in the above "snug fit" scenario, this is 16; in the 6% wastage method that MPE uses, the blocking factor is 3 (3 records to 1 sector); in the (ugh!) 69% wastage at 1 record to 1 sector, the blocking factor is 1. The BLOCK therefore, is BLOCKING FACTOR records — i.e., when the blocking factor is 16, the block is 16*40 = 640 words = 5 sectors.

In general, MPE chooses the blocking factor as follows. If the record size of a file is less than one sector (128 words), the blocking factor = 128/recordsize = the number of records that will fit into one sector; if the record size of a file is greater than 128 words, the blocking factor is always 1. A good example of the possible wastage is when a record is 65 words long; then, 128/65 = blocking factor of 1, wasting 63 words for every 65 words used — a wastage of 49%! If that record was, however, 64 words long, then the blocking factor would be 2, with NO wastage.

By the way, it happens that the blocking factor for a new file can be defined in a :BUILD or :FILE command — always as the second subparameter (between the record size and the F, V, or U record format) of the REC= keyword. Thus, if you want to eliminate the 6% waste due to the blocking factor of 3 on EDITOR keep files, just execute an equation of the form ":FILE filename;REC=,16" right before keeping the file as "filename," and presto! out comes a file with a blocking factor of 16. For already existing files, some disgustingly complicated tricks can be used — or, if you are blessed with a copy of MPEX/3000, just use the BLKFACT= keyword of the %ALTFILE command.

Now, you may wonder, what leads MPE to choose a default blocking factor calculation system that leads to considerable wastage in perhaps one of the most common forms of files? Well, for one, it would be unfair not to remark at this point that the "NO wastage" schemes described above really DO waste some space (although not a lot). The reason for this is that a file (in fact, each extent of a file) must be an integral number of blocks. If it isn't, a full block is allocated for less than "BLOCKING FACTOR" records. Thus, if you have a file containing 50 80-byte records with a blocking factor of 16, it would use up 4 blocks, the last one having only 2 actual records — this file will thus use 21 sectors; however, if that file is built with a blocking factor of 3, it would use up 17 blocks (the last one also having only 2 records), and would thus use only 18 sectors of disc space. However, this consideration is less important for larger files. Another reason for MPE's default blocking factor strategy is that the block and the blocking factor govern more than just disc space usage — they also control certain parameters of buffered file access (see the chapter on FILE ACCESS). However, for most files (especially large ones!) it is beneficial to select your own blocking factors (with the use of the contributed BLOCK program, for instance).

The Variable Record Length File

Let us take a hypothetical EDITOR COBOL-format file. At the beginning of each line there is a 6-digit line number; the other 74 characters contain the line, blank-padded. Now, those trailing blanks, especially in large source files, convey absolutely no information to anybody, and (since the average length of a line could be estimated at half of 74 characters) will cause a wastage of APPROXIMATELY 50% OF THE DISC SPACE USED BY THOSE FILES! But, you reply, if EDITOR built the file with a record length of, say, 40 characters, all of my lines that are longer than 40 characters will get truncated. Well, you're right — but that is not what is to be done! Wouldn't it be nice if EDITOR and/or the file system allowed you to have files not with a FIXED record length, but with a VARIABLE record length — i.e., lines that are 74 characters long will use 74 characters and lines that are 10 characters long will use 10 characters? Well, it does!

In fact, if you type in the little-known /SET VARIABLE command in EDITOR, it will instruct EDITOR to keep the workfile as a variable length record file (WARNING: USE THIS ONLY FOR COBOL AND DATA FILES, NEVER FOR NUMBERED FORTRAN OR SPL SOURCES, OR THOSE SOURCES WON'T BE COMPILER READABLE!!!), thus letting it ignore those trailing blanks, but still keep the file format transparent to other programs that read these files — for example, compilers. In your own programs (not just in EDITOR), you can read variable record length files without changing your programs at all — COBOL's or FORTRAN's READ command can read variable record length files. You can write them without any changes either — if you write a 10-character record to a fixed record length file of 80 characters, the record will be padded with 70 blanks or nulls; if you write that record to a variable record length file, the record will not be padded by anything, thus saving the space required for the padding. To build a variable record length file, specify the third subparameter of the REC= parameter of the :FILE or :BUILD command as "V" (e.g., REC=-80,,V). The record size specified is now no longer the actual record size of each record but rather the maximum; whether the file is ASCII or BINARY now really doesn't matter. Also, do not call in the National Guard (or PICS) when you see on a :LISTF that the END OF FILE for that file is GREATER THAN ITS FILE LIMIT — it can happen with variable record length files.

Therefore, with COBOL source files (especially) and unnumbered data files, variable-length records are usually the way to go; again, however, we must warn you that numbered default-format (e.g., SPL or FORTRAN source) files SHOULD NEVER BE KEPT WITH THE /SET VARIABLE OPTION SET or else they will not be readable by the compiler.

However, as the old proverb says, "EVERY SILVER LINING COMES WITH A CLOUD ATTACHED TO IT," variable record length files have some drawbacks. For one, they can not be accessed directly (for instance, with the FREADDIR, FWRITEDIR, or FPOINT intrinsics, or FORTRAN's READ/WRITE (fnum @ record) construct); i.e., you can read their records sequentially, but you can not ask to get, for instance, the 17th record of the file. Moreover, they cannot be accessed by many file copiers using the fast MR NOBUF file access method (see under FILE ACCESS in this paper), such as HP's own DSCOPY, MPEX's %FCOPY ,,FAST/DSLINE, MPEX's %ALTFILE, SUPRTOOL/ROBELLE, etc. Also, before MPE IV, append access to variable record length files was not supported; it is supported starting with MPE IV.

Another important consideration to keep in mind when using variable record length files is that when you build a new variable record length file with record size RECSIZE and blocking factor BLKFACT, the resultant block size of the file will be not RECSIZE*BLKFACT (as in fixed record length files), but rather RECSIZE*BLKFACT + (BLKFACT+1)*(2 bytes). Thus, if you build a variable record length file of record size 80 bytes and blocking factor 3, the file will actually have a block size of (80*3+4*2)=248 bytes. However, if the same file is built with a blocking factor of 16, the block size will end up being (80*16+17*2)=1314 bytes, not 1280 bytes! The end result is that AN OPTIMAL BLOCKING FACTOR FOR A FIXED RECORD LENGTH FILE MAY BE FAR FROM OPTIMAL FOR VARIABLE RECORD LENGTH FILES!

Incidentally, MPE IV's new INTER-PROCESS COMMUNICATION features (i.e., Message and Circular files) rely EXCLUSIVELY on variable record length files (q.v. COMMUNICATOR issue 26 — the C MIT).

Undefined Record Length Files

There exists another type of disc file — the undefined record length file. These are rather bizarre specimens which are not intended to be and should not be used as disc files, but are rather supposed to be utilized as tape files and terminal files, which are beyond the scope of this paper. ASCII VS. BINARY FILES When using fixed record length files, it often happens that you may write a 30-character record into a file with a record length of 80. Then, what happens to the other 50 characters of the record? Well, for some files (for instance source files) that contain simple text data, you would typically want to initialize it to spaces because of the nature of the file. If that is what you want, you would build that file (EDITOR will build it that way for you) as an ASCII file. This parameter can be specified as the fourth subparameter of the REC= parameter of the :FILE or :BUILD command, e.g., REC=,,,ASCII. However, for some data files, you may want to pad the records with binary zeroes (nulls). Files built in such a way are called BINARY files, and can be built by specifying the BINARY parameter as the fourth subparameter of the REC= parameter of the :FILE or :BUILD command, for example REC=,,,BINARY. Note that this is usually not necessary as BINARY is the default file mode. Also note that since no record padding is done in variable record length files, the ASCII vs. BINARY distinction is usually irrelevant to them.

The File Code

If you do a :LISTF mode 1 or 2 on a group of files, you may notice that some files have a file code of 0 (blanks), some of PROG, USL, EDTCT, KSA M, PRIV, and assorted numeric codes. These filecodes, for the most part, are merely for the sake of file identification — they have no physical influence on the actual contents of files. If you change the filecode of a file (for example with MPEX/30 00's %ALTFILE filename;CODE= command), the contents of the file will not magically change. However, the filecode is useful for identification purposes — for instance, the MPE loader knows that files of filecode PROG are :RUNable program files, the EDITOR knows that files of code EDTCT are /SET FORMAT=COBOL files, QEDIT/ROBELLE knows that files of code 111 are its files. In fact, you can set up your own file identification system for source or data files — you can build files with a certain file code (via the CODE= parameter of a :FILE or :BUILD command), alter the file code (with MPEX/3000 or by copying the file), and examine the file code (via the :LISTF command or, programmatically, with the FGETINFO intrinsic). Certain tools like MPEX also allow you to LISTF files by file code. An example of this kind of file identification system (recently implemented by us) is to set the file code to be the Julian date of the day on which it was created, or some other important date.

Note that the file code of each file is in reality a number — for example, program files (PROG) have a file code of 1029, but they are listed in a :LISTF output as PROG. Also, KSAM files do not actually have a numeric file code that identifies them as such -- they can in reality have any numeric file code. However, KSAM files which have a file code of 0 (which usually shows up as blanks on a :LISTF listing) will be printed as having code = KSAM. Files that are listed as having file code = PRIV are in reality files that have NEGATIVE file codes (like IMAGE files). Unlike usual files, they can only be accessed by programs running in PRIVILEGED MODE. This is handy, for instance, for IMAGE files, to ensure that an ordinary user can not physically change an IMAGE file without going through the existing IMAGE utilities/intrinsics.

User Labels

It is often desirable or necessary to store information in a file in such a way that it can later be retrieved, but is nonetheless transparent when you read it in an ordinary fashion. The concept of USER LABELS provides this capability. With it, you can write special label records (the maximum number of which is specified at open time, defaults to 0, and can be up to 254) with the FWRITELABEL intrinsic, read them with the FREADLABEL intrinsic, but have them be transparent to any user who reads or writes ordinary records to that file. These labels are used by IMAGE, KSAM, and the message system file (e.g., CATALOG and CICAT). Another advantage of user labels is that you can write user labels when you open the file for read access, can read user labels when you open the file for write access, and can open the file for OUT access (see access modes below) which will else all of the file's records but not its user labels.

Carriage Control Files, Relative I/O Files, Message Files, Circular Files, KSAM Files, IMAGE Files, and Other Monsters That Inhabit the HP3000

This paper will not talk about the above types of files (for want of time, will, and disc space). However, maybe sooner or later you will hear the truth about them, too!

CHAPTER II FILE ACCESS

Once a file is built, it really isn't much good if you can't access it — read it, write it, append to it, etc. In this chapter we will discuss the different methods of accessing files that MPE provides for you.

Buffered File Access

A while back we referred to the concept of the BLOCK. Well, it turns out that the block is more than a convenient way of storing records on disc. In fact, it plays a very important role in the default mode of file access called BUFFERED FILE ACCESS. Let us assume that you are reading a 10,000-record disc file which has a record size of 40 words (80 bytes), a blocking factor of 16, and thus a block size of 640 words. Let us assume that you had to do one disc I/O for each record — this would come up to a total of 10,000 disc I/O s, quite a lot!

So, MPE implemented a rather ingenious idea called file buffering. Each file opened as a buffered file has allocated for it a certain amount (default 2, changable at open time with the FOPEN intrinsic or the BUF= parameter of a :FILE equation) of buffers, each of length equal to the file's block size (in this case 640 words). These buffers are placed in an Extra Data Segment (because extra data segment access is faster than disc access) and accessed there. They are read from or written to disc only when a record that is not in the buffer is requested. Thus, for the file described above, only 10,000/16 disc I/O's = 625 disc I/O's is necessary — a considerable savings! The advantage of having more than 1 buffer is that then you can access, for instance, records 17-32 (in one buffer) and 49-64 (in the other buffer) without necessitating a disc I/O each time you switch from one record range to the other. However, if you then read in record 100, the contents of buffer 1 will be flushed out to disc and buffer 1 will then contain records 97-112. In general, with buffering, one disc I/O is required for every (BLOCKING FACTOR) records — in this case, one disc I/O is needed for each 16 records.

In the discussion above, we advised that you set up blocking factors so that BLOCKING FACTOR * RECORD SIZE be an even multiple of 128; thus, for instance, 16 was chosen for files with records of length 40 words. But, there is more than one way to skin a blocking factor! In fact, since 16 * 40 is a multiple of 128, 32 * 40 certainly is too! Very little disc wastage will result from changing the blocking factor from 16 to 32, but each buffer will now be not 640 words long, but rather 1280 words long, and now only 313 (=10,000/32) disc I/Os will be necessary to read the file! A blocking factor of 64 will require less than 160 I/O's, and so on. This will not necessarily halve the time used by the read, but it sure will decrease it. Of course, the same thing can be said for writing to files. We must, however, point out that memory space will be used much more heavily by files that have large blocking factors. Also, the total size of the buffers must be less than or equal to 8,192 words (or 14,000 words starting with the D MIT version of MPE). Since the default number of buffers is 2, this puts an upper limit of 4,096 words (or 7,000 words starting with the D MIT) on the block size of a file. However, you can increase that maximum to 8,192 words (or 14,000 words starting with the D MIT) by opening the file with 1 buffer (by specifying BUF=1 on a file equation).

Multiple Record Non-Buffered Access (MR NOBUF)

The buffering method described above is rather good, but is still not optimal; first of all, access to the extra data segment in which the buffers are located is faster than disc access, but nonetheless not as fast as access to your own stack. Moreover, as was pointed out above, certain memory usage considerations forbid the buffers from being more than 5K to 10K words, which is also not optimal. Wouldn't it be truly wonderful if one could read not just single records, not just blocks of 16 or so records, but 4,000 words at one shot? Well, one can, through the magic of MR NOBUF, probably the MOST POWERFUL AND FASTEST FILE ACCESS METHOD NOW AVAILABLE! MR stands for Multiple Record I/O (do not confuse this with the Multiple RIN capability, also abbreviated MR), and NOBUF stands for No Buffering (this is a bit of a misnomer — it means that it is you, not MPE who will provide the buffer space needed). Note that MR must be used with NOBUF!

Certain factors to beware of when using MR NOBUF are: for one, this method is rather hard to use with variable record length files. Also, the efficiency of this method is best with a buffer size of 4,096 words. Another factor is that when the block size (BLOCKING FACTOR * RECORD SIZE) is not a multiple of 128 words, MR NOBUF is not that much more efficient than ordinary file access, and simple NOBUF access should be used instead.

By far, the best application of MR NOBUF is with file copying. FCOPY, which uses ordinary buffering methods, is often 10 to 20 times slower than MR NOBUF copiers like HP's own DSCOPY, MPEX/3000s %FCOPY ,,FAST/DSLINE or %ALTFILE commands, SUPRTOOL/ROBELLE, and numerous other programs. However, you can do MR NOBUF reading and writing from your own programs by specifying the MR NOBUF access options when accessing the file (or specifying the MR or NOBUF parameters on the :FILE equation — however, a bug present on some versions of MPE forces you to specify MR in the FOPEN because it ignores the MR :FILE equation parameter; see "ANOTHER MPE FEATURE (BUG)" in SCRUGletter, Jan 1981 Vol 4 #1). This will allow you to read more than one record (always at least one block, however) at a time, and also lets you do direct I/O (e.g., with FREADDIR and/or FWRI TEDIR) on a block number rather than record number basis.

However, reading files MR NOBUF in your own programs is rather hard to do because of many concerns that have to do with doing deblocking of records. Because of this, it is suggested that you either do most of your record selection outside of your program (with SUPRTOOL/ROBELLE, for instance), develop your own MR NOBUF I/O routines that can be easily called from your applications programs, or use David Brown's FAST I/O procedures.

MR NOBUF I/O can therefore really cut down the execution time and CPU time demands of your disc I/O-heavy programs. The only problems with MR NOBUF are that it is hard to apply it to variable record length files and KSAM files and that it may (because of the large in-stack buffers necessary) use up a lot of stack space and much memory space.

The Access Types for Disc Files

In the access options parameter of the FOPEN intrinsic or in the ACC= parameter of the :FILE command you can specify the so-called access type which defines whether a program will read the file, write to the file, do both, or append records to the file. There are 7 legal access types, which can be very useful if used properly. The default access type is IN access. This is read-only access — all attempts at writing records to files opened with IN access will fail with File System Error 40 — OPERATION INCONSISTENT WITH ACCESS TYPE. If you only want to read the file, you should open it with this access type; this will prevent your program accidentally writing over the file; it will work even if somebody else has the file opened in Share or Exclusive Allow Read mode (see the SHARING FILES chapter) and it will also work if the file's security prevent you from doing anything but reading that file.

Another type of access is OUT access. OPENING OLD FILES WITH THIS ACCESS TYPE WILL ERASE THEM! If you do not want that to happen, you should open th e file with OUTKEEP access. However, if you want to erase the file, or the file is new, or you do not care about its old contents anyways, this is the access type that should be specified. Note that you need WRITE access to the file to open it in this mode. OUTKEEP access is useful for opening files to write to them, but NOT DESTROYING THEIR OLD CONTENTS (as OUT access would do). You need WRITE access to the file to open it with this access type.

Often you do not need to write over the old contents of a file — you merely need to add new records to it. In that case, APPEND access is for you — it forces the record pointer to be positioned at the end of the file and only permits you to append records to the file. Another advantage of it is that it requires that you have only APPEND access (not WRITE access) to the file to open it thus. So, if you wish to permit users to only append and not overwrite data in a given file, they should be allowed only APPEND and not WRITE access to this file. For instance, VESOFT's SECURITY/3000 permits APPEND access to its security violation log file, but not WRITE access (so user's can not obliterate the record of their violations).

The above access types permit you either to READ ONLY or WRITE ONLY, but never both. INOUT access lets you both READ and WRITE to the file. All intrinsics (except FUPDATE) can be used against that file in this mode. Note that you need READ and WRITE access to open a file in this way.

There is also a special form of access called UPDATE access that is PRECISELY the same as INOUT access except that it permits the usage of the FUPDATE intrinsic. Since this is apparently no less expensive than INOUT access, and requires no extra access to the file, it is suggested that this option be used instead of the INOUT access type because it is more powerful and no more dangerous.

Another access, permissible only to programs that run in Privileged Mode (Ohmigod!), is EXECUTE access; its advantages are twofold. For one, it requires only EXECUTE access to a file, not READ access; moreover, it allows you to write to loaded program or SL files. This is listed only for the sake of completeness, and all you nice non-privileged users out there don't even need to know about it. For a discussion of privileged mode, see PRIVILEGED MODE: USE VS. ABUSE, SCRUGletter July 1981, Vol 4, #4.

Posting End of File to Disc

As was mentioned before, each file has a special record called "the file label" (which contains all sorts of information about the file, such as its type, name, and, among other things, its end of file, which is the number of records which the file contains). Now, if every time that you wrote a record to the file, MPE would have updated that file's file label, your programs would run quite slow — after all, that would mean extra disc I/Os to handle. For this reason, MPE does not post the end of file to disc until a record write would cause it to allocate a new extent (in which case it would have to change the file label anyway), thus saving the extra I/Os.

This is all fine and dandy, provided that MPE will actually get a chance to post the end of file to disc sometime. But, what if the system crashes after you wrote the record but before MPE posted the end of file? Then, even though the record (or records, as the case may be), are already out on disc, MPE does not know about it because the end of file pointer does not reflect this. So, you've just lost all those records that were written before the system crashed. You can, however, minimize your losses through a little-known feature of the file system by calling the FCONTROL intrinsic (see System Intrinsics Manual) with a parameter 6 (WRITE END OF FILE) which lets you post the end of file to disc. If you do this after you write each record, the most records that you will ever lose due to a system failure is one! Of course, this will triple the number of disc I/Os that you'll have to do, so this is not advised for large batch runs; however, if you are updating a disc file interactively, the time it takes to input all of the data from the screen will dwarf the time it will take to do the extra I/O to such a degree that that the posting of the end of file will be virtually free in terms of time, and may save you hours of re-entering vital data.

When You Are Not Alone

When you use any of the access types listed above except read only (IN) access, the file specified will be opened EXCLUSIVELY; that is, you can not open it if anybody else has it opened, but, once you have it opened, NOBODY ELSE CAN USE IT UNTIL YOU CLOSE IT. This is, of course, somewhat of a problem if that file is intended to be read and written by many different users. There are several ways to get around this dilemma.

Exclusive Allow Read Access

One of those ways is Exclusive Allow Read (EAR) access. This permits you to forbid all other users from writing to a file, while letting them read that file. Also, this access (unlike EXCLUSIVE) access will be granted to you even if the file is already being accessed for read access (but not for write access) by someone else. This can be specified by setting the appropriate bits in the access options of the FOPEN call, or issuing a FILE equation with the EAR keyword. TRUE SHARED ACCESS But, you sometimes want not just to have one writer of a file, or one writer and several readers, but MORE THAN ONE PERSON WRITING TO A GIVEN FILE. This can be accomplished with SHARED ACCESS (to use, specify the appropriate bits in the FOPEN call or append the SHR keyword to the file equation for that file), which is the default mode for read only access, but has to be explicitly specified and handled when writing to a file. Shared access is a very complicated form of access, one at which we will look closer in the next chapter.

CHAPTER III SHARED FILES

Sharing Files With Input/Output Access

Merely specifying SHR access when opening the file will get you where you are going — it will allow you and anybody else who opens the file with SHR access to read and write to this file. But, let us suppose the following situation: two processes have opened one file for IN/OUT access in SHR mode, and the following happens:

PROCESS A	PROCESS B
Reads a record	
	Reads the same record
Changes the record	
	Changes the record
Writes the record back	
	Writes the record back

In the above scenario, process A reads the record before process B reads the same record but writes it back out after process B reads it in! That way, process A's changes WILL NOT BE REFLECTED IN THE FILE because of the interference of process B. In fact, what is needed is a method of "LOCKING OUT" all other writers of the file while the file is being updated! Well, MPE's FLOCK and FUNLOCK intrinsics provide this method.

Dynamic Locking and Unlocking for Shared Files

In order to use dynamic locking, the process that opens the file must open it with dynamic locking enabled (the LOCK parameter on the :FILE equation — together with the IN/OUT SHR access, the file equation would now look like :FILE file;LOCK;SHR;ACC=INOUT — or the appropriate bit in the access options of the FOPEN intrinsic call). Then, before each "logical transaction" (a period in time in which the data in the file is not consistent — in the above example, while the record is being changed, the current state of the file does not reflect the true intended state; therefore, the file must be locked before the read and unlocked after the write) the file must be locked and then be unlocked after the end of the transaction (note that opening a file with dynamic locking enabled does not actually lock the file). This will ensure that there will be no inconsistencies like the one shown above. Note that THIS WILL WORK ONLY IF ALL WRITERS LOCK THE FILE IN APPROPRIATE CASES — this locking arrangement only works for programs that honor it.

Sharing Files with Append Access

In some cases, however, locking does not really help. For example, if two writers are just writing to a file (no reading, etc.), the "logical transactions" like the ones described above are composed of merely one write. For these transactions, it does no good to lock the file. One of the most common example of this type of file access is shared append access to a file by two or more writers.

In fact, if the file has a blocking factor of 1, there is no need to do anything but the write. However, look at what happens when the file has a blocking factor other than 1, for example 3; consider process A and process B, both writing to the same file:

PROCESS A	PROCESS B
Record 1 written; kept in buffer	
	Record 1 written; kept in buffer
Record 2 written; kept in buffer	
	Record 2 written; kept in buffer
Record 3 written; buffer flushed	
	Record 3 written; buffer flushed

Note that, by the principles of buffering, the actual disc I/O is not done until the third record is written and the buffer is flushed out to disc. But, because of that, when it is flushed out to disc, the buffers from process A and process B interfere with each other, and data can be lost. Therefore, the rule for locking when appending (or performing any other such operation in which each transaction contains only one operation) is: LOCK WHEN THE BLOCKING FACTOR IS GREATER THAN 1; IF THE BLOCKING FACTOR IS 1, LOCKING I S UNNECESSARY.

Multiple File Access

Another way to ensure that no data is lost while writing to a file is with a useful tool (which is even more useful under MPE IV) called MULTIPLE FILE ACCESS. With multiple file access in shared mode, the internal file control information and the I/O buffers are shared, as well as the file itself, thus avoiding many problems of ordinary shared access.

So, if process A and process B (IN THE SAME JOB/SESSION) access a file SHARED, APPEND, and MULTI, then their internal end of file and buffer pointers are shared; thus, the risk of one's file I/O interfering with the other's is eliminated. To specify MULTI-access, set the appropriate bit in the FOPEN parameters or specify the ;MULTI keyword on a :FILE command for the file in question.

So, very many of the problems and complicated locking strategies discussed above can be avoided if MULTI-access to that file is used. However, there are two things that you must keep in mind when using MULTI-access; for one, ordinary sequential reads and writes to that file will not behave as expected. Why? Well, the current record pointer is among those values that is shared with MULTI-access and thus if process A reads a record sequentially and then process B requests to read a record sequentially, process B will get the next record because the record pointer was already incremented by A's read. Thus, if the two processes read the file sequentially with MULTI-access, each one will read approximately half the file instead of the full file!

MPE III vs. MPE IV

Another problem for all you unlucky people who still do not have MPE IV, MULTI-ACCESS IS PERMISSIBLE ONLY WITHIN ONE JOB/SESSION UNDER MPE III! However, under MPE IV, you can use the GMULTI (Global MULTI access), which can be specified in the FOPEN parameters or with the GMULTI keyword of the :FILE equation, to have MULTI-ACCESS ACROSS JOBS/SESSIONS, with which you can avoid most of the problems of shared file access very easily.

More About Locking

There are two methods of locking files: UNCONDITIONAL, which means "if the file is already locked by somebody else, wait for them to unlock it, and then establish the lock" and CONDITIONAL, which means "if the file is already locked by somebody else, return to me immediately with an error condition." The UNCONDITIONAL method is usually the most useful, although the CONDITIONAL option is handy when you do not want to take the risk of waiting a long time (if the program that has it locked won't unlock it for a while). Needless to say, the file should not be locked for a long time, and SHOULD NEVER BE LOCKED WHILE A TERMINAL READ IS GOING ON unless you do not mind the fact that if the terminal operator goes to lunch, everybody else who tries to unconditionally lock the file will hang.

Locking Multiple Files, Or The Secrets of Multiple Rins (MR)

Let us consider another hypothetical circumstance: Process A locks File 1; meanwhile Process B locks File 2. Then, Process A tries to unconditionally lock File 2 and is then impeded until Process B unlocks File 2. Meanwhile, Process B tries to unconditionally lock File 1 and is then impeded until Process A unlocks File 1. Thus, Process A is waiting for Process B and Process B is waiting for Process A. Result: Deadlock. Both processes are hung until the system is re-started. The sages of Cupertino thought of that when designing the system; in fact, their solution (which may not sound like much of a solution, but is better than nothing) is TO FORBID PROGRAMS TO LOCK MORE THAN ONE FILE AT A TIME. But, one may object, what if I have to lock more than one file at a time? Well, the answer to that problem is that you can get around (but at your own risk) that restriction if the program that does the locking has Multiple RINs (MR — not to be confused with Multiple Record access) capability (i.e., was :PREPped with it. By the way, RIN stands for Resource Identification Number. These programs can, IF THEY REALLY HAVE TO, lock two or more files at a single time. Needless to say, this capability should not be freely given to everybody and his brother, but only to people who really need it, and smart enough to use it without causing deadlocks.

That brings us to the problem of: How do you get around the deadlock problem? Well, you may have already noted that the reason why the programs got into a deadlock was that one locked File 1 before File 2 and the other locked File 2 before File 1. If they had only kept a consistent locking arrangement (e.g., File 1 must ALWAYS be locked before File 2), they would not have had the problem — this is probably the best way to avoid the deadlocks. Another way is to lock the files CONDITIONALLY, and if the lock fails, do something else (or even go into a loop, which can at least be broken out of by aborting the job or doing a break/:ABORT, rather than re-starting the system).

Summary of Locking And Locking Strategy

The following are the 10 commandments of locking:

1. *Thou shalt lock around logical transactions which involve two or more operations.* For example, that kind of a logical transaction would be a read of a record followed by a modification of that record followed by a write. If you do not lock around this, you stand the risk of losing data consistency.

2. *Thou shalt also lock around all logical transactions that involve a file which you share with somebody who has transactions which involve two or more operations.* That means that if process A's transactions are just single writes and process B's transactions are reads followed by writes, both process A AND process B must lock around their transactions.

3. *Thou need not lock a shared file if all its writers' transactions involve just one operation and its blocking factor is 1.* Thus, if process A and process B are writing to a shared file, and their transactions are merely single writes (e.g., they are appending to a file), neither one has to lock the file.

4. *Thou shalt use GMULTI access under MPE IV when you are appending to a shared file.* This can save you time, worry, and your neck.

5. *Honor thy locking arrangements.* This means that if it has been decided that a shared file is to be locked by its writers, all writers must lock it. If so much as one writer fails to lock the file, all of the locking arrangements will be useless.

6. *Thou shalt not keep a file locked while a terminal read is in progress.* If you did, then the file will be locked down until something is entered, which could mean an indefinite waiting period for any other program that wants to lock the file.

7. *Thou shalt not lock more than one file at the same time without MR Capability.* The second file lock will fail unless your program was :PREPped with MR capability.

8. *Thou shalt protect thyself from deadlocks by establishing a fixed file locking sequence if you use MR capability.* Thus, if process A locks file 1 and then file 2, process B must lock in the same order, i.e., file 1 and then file 2 (not file 2 and then file 1!).

9. *Thou shalt not give MR capability to just anybody.* MR capability can cause big trouble, and thus should be passed out sparingly.

10. *Thou shalt use IMAGE/3000 if thy file locking arrangements get too complicated.* IMAGE/3000 has file locking capabilities far superior to MPE's file locking features. If you find that your locking arrangements are getting too complicated or programs are waiting inordinate amounts of time to get at a shared file, think about converting it to an IMAGE file — it may be worth your while.

CHAPTER IV FILE DOMAINS AND EQUATIONS

Permanent and Temporary Files

Most of the files that we discussed in previous sections were usual PERMANENT files — files that, once built, exist until they are :PURGEd or somehow deleted. There is, however, another type of file, one that is also often quite useful. This is the JOB/SESSION TEMPORARY FILES. These files, once built (by placing the ;TEMP keyword on the :BUILD or :FILE command), exist until they are :PURGEd (by performing a ":PURGE filename,TEMP") OR UNTIL THE JOB OR SESSION IN WHICH THEY WERE CREATED LOGS OFF. Why are these files desirable? Imagine, for instance, that you want to create a certain file that you want to stream. After the file is streamed (in the same job or session that it was built in), you no longer need it. If you were to create that file as a permanent file and then purge it, it is quite possible that somebody else may have built a file with the same name; for instance, if the same program is being run on another terminal and that file is created there.

However, if you create it as a temporary file, you can be certain that creating it will not interfere with anybody else; the nature of job/session temporary files is such that two different jobs or sessions can create within them temporary files with the same name which do not interfere with each other.

Most MPE commands either attempt to open the file given to them as a temporary file and then (if the temporary open fails) as a permanent file (e.g., :STREAM,:COBOL,:RUN, etc.), thus being able to accept both temporary and permanent files, or have special keywords that instruct them to open the file as a temporary file (e.g., PURGE file,TEMP). Programs that open files as permanent can be instructed to open the file as job/session temporary by issuing a file equation of the form ":FILE fil ename,OLDTEMP". Note that some commands and subsystems (e.g., :BASICOMP, :PREP, :SEGMENTER's -BUILDUSL command) build files as temporary files; others can be instructed to build files as temporary by using a file equation like ":FILE filename;TEMP".

If you need to keep a temporary file as a permanent file with the same name, you can do a ":SAVE fil ename"; if you want to keep it as one with a different name, do a ":RENAME oldfile,newfile,TEMP" and a ":SAVE newfile". The names of your temporary files can be listed with LISTEQ2 or (in a more complete, :LISTF-like format, with MPEX/3000's %LISTF fileset: TEMP command).

$NEWPASS and $OLDPASS

Two other useful critters are the system-defined files called $NEWPASS and $OLDPASS. Consider, for instance, the :COBOL command. When the USL file is omitted on this command, it is usually followed by a :PREP command that is to prepare the resultant USL file into a program file. But, what intermediate USL file should be used? Well, if you use a permanent or temporary file you run the risk of having a file with that name already in existence. This is where $NEWPASS and $OLDPASS come in. $NEWPASS is a peculiar file that, when closed, magically turns into $OLDPASS. So, once you open $NEW PASS, write to it, and close it, you can then open $OLDPASS, and read it.

So, in the case of the :COBOL and :PREP, the USL file parameter of the :COBOL command defaults to $NEWPASS. The USL file is closed, and, presto!, it becomes $OLDPASS. Now, you can execute a command of the form ":PREP $OLDPASS,progfile", and that USL will be :PREPed into the specified program file. If you really want to be fancy and you don't need the program file to be a temporary or permanent file, you can do a ":PREP $OLDPASS,$NEWPASS", and, after this is done, the program file (which was specified as $NEWPASS) becomes $OLDPASS. Now, you can just ":RUN $OLDPASS". Note that $OLDPASS contains the USL file from :COBOLPREP (or :FORTPREP, :SPLPREP , etc.) and the program file from :COBOLGO (or :FORTGO, :SPLGO, etc.).

If you decide that you want to save the contents of $OLDPASS in a permanent file, just do a ":SAVE $OLDPASS,filename". A rather bizarre undocumented feature is that to save $OLDPASS as a TEMPORARY file, you can do a ":RENAME $OLDPASS,filename"! Of course, $OLDPASS vanishes as soon as you :BYE off.

The Care and Feeding of :File Equations

Perhaps one of the single most important and least understood tools in handling files is the :FILE equation. The file equation allows one to re-define certain open parameters of old and new files. For example, let us say that you are keeping a file with EDITOR, and you want to keep it with blocking factor 16 and 32 extents. Then, you would issue the file equation ":FILE filename;REC=,16;DEV=,32". Note that THIS DOES NOT BUILD THE FILE! However, when you execute the /KEEP command (and EDITOR therefore opens the file) or when you open it from your own or any other program as a new file, it will be opened with blocking factor 16 and 32 extents.

If, however, the specified file already exists and has a blocking factor of 3 and 8 extents and you issue the file equation in hopes that the equation will magically transform it, you're in for a letdown. This is because if that file already has a blocking factor of 3, it will always have a blocking factor of 3 even if you say on the :FILE equation or when opening the file that it has a blocking factor of 16. Its blocking factor is 3 and merely opening it with another blocking factor changes nothing. To truly change the blocking factor, record size, number of extents, file limit, or any one of the other file parameters, you need to either rebuild the file (remember, these parameters can be redefined when you are building a new file) and copy the old contents of the file into it, or use utilities such as MPEX/3000.

However, some options can be redefined for OLD files. These are not the file options (like CCTL or REC) but the access options (like ACC, BUF, MR, etc.), which are not inherent parts of the file, but rather attributes of the access, defined when the file is opened. These can therefore be redefined for OLD or NEW files. Another class of :FILE equation parameters governs actions that are to be performed not at OPEN time, but rather at CLOSE time. The only parameters in this class are disposition parameters. The SAVE option instructs the program to close the file as a permanent file; the TEMP option tells it to close the file as a job/session temporary file (q.v. TEMPORARY vs. PERMANENT FILES); and, the DEL option will delete the file referenced when it is closed. Note that although all :FILE equation parameters correspond to some FOPEN or FCLOSE parameter, not all FOPEN and FCLOSE parameters can be redefined with a :FILE equation; for instance, the number of user labels (on open), or the flag that indicates whether space between end of file and file limit is to be released (on close) can not be redefined with :FILE equations.

If you do not want the user to be able to re-define the open or close parameters of a file, you should open the file with the Disallow File Equations bit in the FOPTIONS parameter of the FOPEN intrinsic set.

APPENDIX B

A Glossary of Common Disc File Handling Terms

ACCESS-MODES — The file's ACCESS MODE (one of IN, OUT, OUTKEEP, APPEND, INOUT, UPDATE, or execute) that is defined at file open time and restricts the actions that can be performed on the file. This can be redefined with the ACC= parameter of the :FILE equation. See APPEND ACCESS, IN ACCESS, INOUT ACCESS, OUT ACCESS, OUTKEEP ACCESS, UPDATE ACCESS.

ACCESS-OPTION — The ACCESS OPTIONS are a parameter to the FOPEN intrinsic (q.v.) that define the access mode, sharing status, dynamic locking flags, etc. See FOPEN.

ASCII — ASCII files are fixed/undefined length files that are padded or initialized to blanks instead of zeroes. That is, writing a record that is shorter than the record size causes the result to be blank-padded. To create, use ASCII as the 4th subparameter of the REC= parameter of the :FILE/:BUILD command. See BINARY.

BINARY — BINARY files are fixed/undefined length files that are padded or initialized to zeroes (nulls). To create, use BINARY as the 4th subparameter of the REC= parameter of the :FILE/:BUILD command. See ASCII.

BLOCK — A BLOCK is the unit in which data is transferred between I/O devices and file buffers on disk. 1 BLOCK = BLOCKIN G FACTOR records. Each block always starts on a sector boundary, and thus, for disc space usage efficiency should be equal to an integral number of sectors whenever possible. See BLOCKING FACTOR, BLOCK SIZE.

BLOCKING-FACT — The BLOCKING FACTOR is the number of records per block. To optimizing disc space usage, set the blocking factor such that BLOCKING FACTOR * RECORD SIZE is a multiple of 128 words. To optimize file access speed, set the blocking factor as large as possible. To minimize memory usage, set the blocking factor as small as possible. To set the BLOCKING FACTOR for new files, specify it as the 2nd subparameter of the REC= parameter of the :FILE or :BUILD command. See BLOCK, BLOCK SIZE.

BLOCK-SIZE — For fixed record length files, BLOCK SIZE = BLOCK FACTOR * RECORD SIZE. For variable record length files, BLOCK SIZE = BLOCK FACTOR * RECORD SIZE + (BLOCK FACTOR + 1) * (2 bytes). The most efficient disc space usage occurs when the block size of a file is equal to an integral number of SECTORS. See BLOCK, BLOCK FACTOR.

BUFFERING — The default mode of file access is BUFFERED FILE ACCESS — in this mode records are not immediately read from or written to disc, but rather kept in an extra data segment which contains (BUFFERS) buffers of length (BLOCK SIZE) words each. See BUFFERS, NOBUF ACCESS.

BUFFERS — When a file is accessed in buffering mode, a certain number of BUFFERS is allocated, each one of length (BLOCK SIZE) words, in one extra data segment. The default number of buffers is 2, and can be redefined with the BUF= parameter of a file equation. See BUFFERING.

DEADLOCK — A situation in which two processes are hung, each one waiting for the other to do something. This can happen when several files are locked by processes with MR capability. See LOCKING FILES, MR CAPABILITY.

DEVICE — The DEVICE on which a disc file resides can be a single disc (specified by placing its device number in the FOPEN call or as the 1st subparameter of the DEV= keyword of the :FILE equation) or a device class, a collection of disc devices grouped under a generic name (specified in the same place as the device number). All of the extents of the file are placed on this device or device class.

DOMAIN — The DOMAIN of a file can be PERMANENT or TEMPORARY. This can be specified on a :BUILD command (;TEMP indicates TEMPORARY, omission of it means PERMANENT) or a :FILE command (for old files, :FILE filename,OLD means PERMANENT and :FILE filename,OL DTEMP means TEMPORARY; for new files, :FILE filename;TEMP means TEMPORARY and ;SAVE means permanent). See PERMANENT, TEMPORARY.

EAR — EAR (short for Exclusive Allow Read) is an access mode that permits a user to open a file for write access, but still allow other users read access to the file. See EXCLUSIVE ACCESS, SHARED ACCESS.

END-OF-FILE — The END OF FILE is usually the number of records that have been written to a given file. It is usually less than the file limit (q.v.), which is the maximum number of records in a file, but could be greater than it in variable record length files (q.v.).

EXCLUSIVE — EXCLUSIVE ACCESS to a file is an access mode in which the accessor forbids everybody else to access that file while he is accessing it. This mode is the default mode for all non-read access. It can be specified in the access options of an FOPEN call or in a :FILE equation with the EXC parameter. See EAR ACCESS, LOCKING, SHARED ACCESS.

EXTENT — An EXTENT is a collection of blocks that occupies contiguous space on a given disk. There can be up to 32 such extents in a file, but the default is 8. See EXTENT SIZE, MAXIMUM EXTENTS, NUMBER OF EXTENTS.

EXTENT-SIZE — The extents of any file must all be of equal length (in sectors), except the last one, which may be of smaller length. For formulae for these lengths, see APPENDIX B — DETERMINING DISC SPACE USAGE. See EXTENT.

FILE-CODE — The FILE CODE of a file is an integer which describes the type of this file; some of the more common codes have special mnemonics corresponding to them (e.g., PROG = 1029 = file code of program files). These mnemonics show up on :LISTFs of that file, and can also be specified on a :BUILD or :FILE command. The code, whether mnemonic or numeric, can be placed on the CO DE= parameter of a :BUILD or :FILE command.

FILE-EQUATION — A file equation is a useful tool that allows a user to redefine certain open or close parameters of the file (e.g., the file code (CODE), the access type (ACC), the close disposition (SAVE/TEMP), etc.). It can be specified through the MPE :FILE command.

FILE-LABEL — The FILE LABEL of a file contains information about that file (e.g., file name, creator id, file code, record size, extent information, etc.) needed by MPE. Ordinary users need not worry about this entity.

FILE-LIMIT — The maximum number of records permitted in a file, necessary for knowing how much disc space to allocate, specified at file creation time. Note that the END OF FILE can actually exceed the FILE LIMIT for variable record length files. The file limit can be specified in a :BUILD or :FILE command as the first subparameter of the DISC= keyword.

FIXED-LENGTH — FIXED RECORD LENGTH files are files whose records have a fixed length — if a record of smaller length is written to the file, the record is padded on the right with an appropriate number of blanks (ASCII files) or nulls (BINARY files). An example of this kind of file is the usual EDITOR file which has a fixed length of 80 bytes. To build fixed record length files, specify F as the third subparameter of the REC= parameter on a :BUILD or :FILE command (e.g., REC=-80,,F). See VARIABLE RECORD LENGTH, UNDEFINED RECORD LENGTH.

FOPEN — FOPEN is a system intrinsic that permits its caller to open a file. BASIC, COBOL, FORTRAN, and RPG users need not be concerned about this intrinsic because their languages provide file access features already (this is therefore mostly used by SPL programmers); however, we often allude to this intrinsic in this paper because all file open commands in all languages eventually translate out to this intrinsic.

GMULTI-ACCESS — GMULTI access is an extended form of MULTI access (q.v.) available only on MPE IV. Its usage (which can be specified by appending the GMULTI keyword to the :FILE equation) together with SHR and ACC=APPEND provides a painless way of appending to shared files. See MULTI, SHARED ACCESS.

LOCKING — MPE's DYNAMIC FILE LOCKING mechanism (available through the FLOCK and FUNLOCK intrinsics) gives users a way to have more than one program write to a file without jeopardizing data consistency. In order to call FLOCK and FUNLOCK, the file must have been previously opened with the dynamic locking access option set (which can be done in the FOPEN call or using the LOCK parameter of the :FILE command). See DEADLOCKS, MULTIPLE RINS, SHARED ACCESS.

MAX-EXTENTS — The MAXIMUM NUMBER OF EXTEN TS defines into how many extents (q.v.) a file is to be split. Note that (usually) not all of these extents are allocated at the time a file is built — the default is 1 (although more can be allocated initially by specifying their number as the 3rd subparameter of the DISC= keyword of the :FILE command). The maximum number of extents can be specified as the 2nd subparameter of the DISC= keyword of the :FILE command, and defaults to 8. See EXTENTS, NUM EXTENTS.

MULTI-ACCESS — MULTI access is a form of access that is very useful for sharing files. It permits you to share not just the files but also internal file control information and file buffers. It can be specified by placing the MULTI keyword on a :FILE command. See GMULTI ACCESS, SHARED ACCESS.

MULTIPLE-RINS — The MR (MULTIPLE RINS) capability is an account, file, group, and user capability that governs a program's ability to have more than one file locked at at a time. In order for a program to be permitted to do this, it must have been :PREPped with MR capability by a user who had MR capability, and it must reside in a group that has MR capability. See DEADLOCKS, LOCKING.

MULTI-RECORD — MULTI-RECORD ACCESS (abbreviated MR) is a mode in which a file accessor can read more than one record at a time, thus greatly speeding up file access. This option must be used together with the NOBUF option (see NOBUF ACCESS). It can be specified on a :FILE equation as the MR parameter. See NOBUF ACCESS.

$NEWPASS — $NEWPASS is a special system-defined temporary file that, when closed, turns into $OLDPASS (q.v.). This file (and $OLDPASS) disappear (along with all job/ session temporary files) at logoff time. See $OLDPASS, TEMPORARY FILES.

NUM-BUFFERS — The NUMBER OF BUFFERS is the number of I/O buffers allocated for buffered file access (q.v.). This number can be specified with the BUF= parameter of a :FILE equation. See BUFFERING.

NUM-EXTENTS — The NUMBER OF EXTENTS is the number of extents that that are currently allocated in the file; this starts out as the initially allocated number of extents (see EXTENTS), and is increased by 1 whenever a record is written to the file which will not fit into the currently allocated number of extents. See EXTENTS, MAXIMUM EXTENTS.

$OLDPASS — $OLDPASS is a special system-defined temporary file that was the last $NEWPASS (q.v.) file closed. This file disappears at logoff time, but can be saved with the MPE :SAVE command. See $NEWPASS, TEMPORARY FILES.

PERMANENT-FILE — A permanent file is a disc file that is accessible by all users in the system (that have the proper access to it, of course) and remains until it is :PURGEd, as opposed to a temporary file (q.v.) that can be accessed only by the session in which it was created and is automatically deleted when that session logs off. The fact that a file is to be accessed as an OLD permanent file can be specified by executing a file equation of the form ":FILE filename,OLD"; the fact that a file is to be saved as a NEW permanent file can be specified by placing the SAVE keyword on a :FILE equation for that file. See TEMPORARY FILES.

RECORD-LENGTH — The RECORD LENGTH of a file is the length of each records in that file if it is a fixed or undefined record length file, and the maximum length of the records in that file if it is a variable record length file. This parameter can be specified as the 1 st subparameter of the REC= parameter on a file equation. See FIXED RECORD LENGTH, UNDEFINED RECORD LENGTH, VARIABLE RECORD LENGTH.

SECTOR — A SECTOR is 128 words of disc space.

SHARED-ACCESS — Files open in SHARED ACCESS mode can be written by more than one program at the same time. This option can be specified in a :FILE equation with the SHR parameter. It is imperative for data consistency that the dynamic locking (q.v.) facility be used by all programs that write to a file shared by two or more writing programs. See EAR ACCESS , EXCLUSIVE ACCESS.

TEMPORARY-FILE — A temporary file is a disc file that can be accessed only by the session that created it, and is automatically purged when that session logs off. It can, however, be saved as a permanent file (q.v.) with the MPE :SAVE command, and purged before the session logs off with the PURGE filename,TEMP command. The fact that a file is an OLD temporary file can be specified by using a file equation like ":FILE filename,OLDTEMP"; the fact that it is to be saved as a NEW temporary file can be specified by appending the TEMP keyword to a :FILE equation for that file. See PERMANENT FILES.

UNDEFINED-LEN — Undefined record length files are not intended to be used as disc files. Use instead fixed / variable record length files. See FIXED RECORD LENGTH, VARIABLE RECORD LENGTH.

USER-LABELS — User labels are records which, although they are parts of the file, are transparent to the normal reader of that file, and can only be accessed via the FREADLABEL and FWRITELABEL intrinsics.

VARIABLE-LEN — Variable record length files are files in which not all records have to have the same length. When records of length less than the record length (which, incidentally, is the maximum length of any record in that file) are written to the file, no padding is done (which means that the ASCII / BINARY distinction has no meaning here), but rather the size of the record to be written becomes the record length of that particular record. A result of this is that no space is wasted due to padding, which makes these files much more efficient users of disc space than fixed record length files (q.v.). See FIXED RECORD LENGTH, UNDEFINED RECORD LENGTH.

APPENDIX B

Determining Disc Space Used By Files Given File Parameters

Perhaps because there are so many different file parameters (record size, blocking factor, end of file, file limit, etc.) that are involved in determined the disc space used up by a certain file, the formula for this calculation is hard to come by and is quite complicated. However, we will attempt to list it together with all its interesting ramifications below. Note that this method will work only for FIXED RECORD LENGTH FILES that are to be WRITTEN IN A SEQUENTIAL FASHION (i.e., no directed writes). The parameters needed for this algorithm are the RECORD SIZE (in words), BLOCKING FACTOR, END OF FILE, FILE LIMIT, NUMBER OF USER LABELS, and MAXIMUM NUMBER OF EXTENTS REQUESTED. This method will yield the NUMBER OF SECTORS USED BY THE FILE, THE EXTENT SIZE OF MOST EXTENTS, THE EXTENT SIZE OF THE LAST EXTENT OF THE FILE, THE MAXIMUM NUMBER OF EXTENTS GRANTED, THE NUMBER OF EXTENTS ACTUALLY USED, and THE BLOCK SIZE OF THE FILE.

Blocking Considerations

The first parameter that must be determined for this calculation is the BLOCK SIZE, in SECTORS, which we will denote by the "variable" name BLKSIZE. Using standard SPL notation, the names BLKFACT = blocking factor and RECSIZE = record size, and keeping in mind that ALL DIVIDES PERFORMED BY US FROM NOW ON WILL BE "CEILING" DIVIDES, i.e., DIVIDES IN WHICH THE RESULT IS THE SMALLEST INTEGER THAT IS LARGER THAN OR EQUAL TO THE FRACTIONAL DIVIDE RESULT (e.g., 5/2=3, 20/4=5), we get the following formula:

```
BLKSIZE:=(RECSIZE*BLKFACT)/128;    << Record size IN WORDS >>
```

Next, we must find out the number of blocks (not records, but blocks) that are used up by the data portion of the file and the label (user label and file label) portion of the file. The formulae for this (note FLIMIT = file limit, ULAB = number of user labels allocated) are:

```
DATABLKS:=FLIMIT/BLKFACT;
LABBLKS:=(ULAB+1)/BLKSIZE;    << the 1 is for the file label >>
TOTALBLKS:=DATABLKS+LABBLKS;   << blocks used by both >>
```

Extent Considerations

At this point, we can determine the extent size (in blocks or in sectors) of each file extent. The formula is (given MAXEXTS is the maximum number of extents requested by the file creator at creation time) as follows:

```
EXTSIZE'BLOCKS:=TOTALBLKS/MAXEXTS;    << in blocks >>, or
EXTSIZE'SECTORS:=EXTSIZE'BLOCKS*BLKSIZE;    << in sectors >>
```

For our purposes, we will use the EXTSIZE'BLOCKS-in-blocks formula. Now, let us digress for a moment. As we have said before, the maximum number of extents of a given file can be specified on a :BUILD or :FILE command, and defaults to 8. But, the maximum number of extents actually granted (this is NOT the number of extents actually used!) may be smaller than the maximum number of extents requested in this way! In order to explain the reason for this, we must first recall a fact that will be of paramount importance to us in this entire discussion: ALL EXTENTS OF A FILE MUST BE OF THE SAME LENGTH, EXCEPT THE LAST ONE, WHICH MAY BE OF SMALLER LENGTH. Let us suppose that you try to :BUILD a file with 100 blocks and 16 as the maximum number of extents (for instance, with an MPE command like :BUILD MYFILE;DISC=100,16). Now the file system must fit an integer number of blocks into one extent. Now, how many blocks can fit into one extent? Well, the number is 7 (the ceiling of 10 0/16). But, by the rule stated above, all extents of a file except the last one must be of the same length. Thus, each extent except the last one must be 7 blocks long. But, only 14 such extents can fit into 100 blocks, leaving 1 2-block extent! So, the file system can not possibly grant you a maximum number of extents larger than 15, even though you asked for 16! n short, the "real" number of maximum extents granted turns out to be:

```
      REALMAXEXTS:=TOTALBLKS/EXTSIZE'BLOCKS;
   where TOTALBLKS and EXTSIZE'BLOCKS were defined above.
```

where TOTALBLKS and EXTSIZE'BLOCKS were defined above.

Now, the above statements have yet to use the END OF FILE parameter. Nevertheless, this parameter is a vital one to our calculation. It permits us to determine another crucial factor, the number of extents currently used (USEDEXTS), through the following formula:

```
USEDEXTS:=(LABBLKS+EOF/BLKFACT)/EXTSIZE'BLOCKS;
```

The above takes the number of blocks actually used by the file and divides it by the number of blocks per extent, thus getting the number of extents actually used. Now, we have the answer: if the number of extents used is the real maximum number of extents, i.e., all of the file's extents are allocated, the number of sectors used can be found by the following:

```
SECTORS:=TOTALBLKS*BLKSIZE;
```

If, however, some of the extents of the file remain unallocated, we can find the number of sectors used with this formula:

```
SECTORS:=IF USEDEXTS=REALMAXEXTS THEN
              TOTALBLKS*BLKSIZE  << if all extents are allocated >>
           ELSE
              USEDEXTS*EXTSIZE'BLOCKS*BLKSIZE;
```

The Facts in a Nutshell

In short, the above rantings and ravings boil down to the following algorithm:

```
(variables:
MAXEXTS  = maximum number of extents requested
RECSIZE  = record size (in words) of the file
BLKFACT  = blocking factor of the file
FLIMIT   = the file's file limit
EOF      = the file's end of file
ULAB     = the number of user labels allocated in that file)
BLKSIZE:=(RECSIZE*BLKFACT)/128;
DATABLKS:=FLIMIT/BLKFACT;
LABBLKS:=(ULAB+1)/BLKSIZE;
TOTALBLKS:=DATABLKS+LABBLKS;
EXTSIZE'BLOCKS:=TOTALBLKS/MAXEXTS;
REALMAXEXTS:=TOTALBLKS/EXTSIZE'BLOCKS;
USEDEXTS:=(LABBLKS+EOF/BLKFACT)/EXTSIZE'BLOCKS;
SECTORS:=IF USEDEXTS=REALMAXEXTS THEN
              TOTALBLKS*BLKSIZF
           ELSE
              USEDEXTS*EXTSIZE'BLOCKS*BLKSIZE;
```

Let us analyze an example case (you can verify it yourself!):

```
MAXEXTS  = 8 extents
RECSIZE  = 40 words
BLKFACT  = 3 records per block
FLIMIT   = file limit of 10000
EOF      = 4600 records
ULAB     = 0 user labels
BLKSIZE              := (40*3)/128 = 1 sector;
DATABLKS             := 10000/3 = 3334 blocks;
LABBLKS              := 1/1 = 1 block;
TOTALBLKS            := 3334+1 = 3335 blocks;
EXTSIZE'BLOCKS       := 3335/8 = 417 blocks;
REALMAXEXTS          := 3335/417 = 8 extents;
USEDEXTS             := (1+4600/3)/417 = 4 extents;
SECTORS              := since USEDEXTS (4) <> REALMAXEXTS (8), then
                        4*417*1 = 1668 sectors;
```

APPENDIX C

A Summary of Methods to Save Disc Space

The following is a summary of some of the possible methods of saving disc space without deleting files or making them unreadable (methods are arranged in order of descending average percentage savings):

% savings	method
25%-75%	Convert source files to QEDIT/ROBELLE format; this format is very efficient in terms of disc space usage yet still readable by compilers.
25%-50%	Convert data/COBOL files to variable record length; this can be accomplished with EDITOR's /SET VARIABLE command.
0%-50%	Improve blocking factor of files; a file's block size should be a multiple of 128 words or disc space will be wasted.
0%-25%	Set file limit of files to end of file; if the file limit of a file is not its end of file disc space is probably being lost. Note that for data files, the file limit should be greater than end of file to allow for expansion.

These operations can be performed on files one by one, or en masse using MPEX/3000.

APPENDIX D

A Summary of Methods to Speed Up File Access

The following is a summary of some possible methods of speeding up disc file access, arranged in order of descending average percentage savings of file access time:

% savings	method
50%-95%	Use MR NOBUF access for file reading/writing; for easy use of this access method, David Brown's FAST I/O routines are suggested.
5%-10%	Increase the block factor of files; this will increase the block size, and thus the buffer size of files accessed with buffering, and thus decrease the number of disc I/Os needed to access them. Make the block size as large as possible, but no more than 8,000. Set BUF=1 (only 1 buffer) to avoid getting file system error 57 (OUT OF VIRTUAL MEMORY).

APPENDIX E

Related Papers / Useful Programs

As we could not (and never intended to) say everything there is to say about disc files, we would like to refer you to the following useful reference documents and utility programs:

PAPERS

"Another MPE Feature (BUG)." A discussion of a bug in Multi-Record file access by Vladimir Volokh, VESOFT Consultants. *SCRUGletter*, Volume 4, Issue 1 for January 1981.

"How to Avoid Problems With MPE Carriage Control (CCTL)." All there is to know (well, almost) about Carriage Control. Robert M. Green, Robelle Consulting Ltd., 27597-32B Avenue, Aldergrove, B.C. Canada V0X 1A0.

HP3000 Computer Systems MPE Commands Reference Manual. Section VI — MANAGING FILES.

HP3000 Computer Systems MPE Intrinsics Reference Manual. Section III — ACCESSING AND ALTERING FILES.

HP3000 Computer Systems MPE IV Intrinsics Reference Manual. Section III — INTERPROCESS COMMUNICATION AND CIRCULAR FILES. Section X — ACCESSING AND ALTERING FILES.

"Privileged Mode — Use and Abuse." What is privileged mode and how to use it safely by Eugene Volokh, VESOFT Consultants. *SCRUGletter*, Volume 4, Issue 4 for June 1981.

SOFTWARE

"FAST I/O (aka BLOCKED I/O)." A product that permits fast, easy MR NOBUF file access available from EASY Software Co., 410 Chipeta Way, Research Park, Salt Lake City, UT 84108.

"MPEX/3000." Many useful extensions to MPE available from VESOFT Consultants.

"QEDIT/ROBELLE." A superior editor, with disc space-saving features available from Robelle Consulting Ltd., 27597-32B Avenue, Aldergrove, B.C. Canada V0X 1A0.

APPENDIX F

Cryptic File System Error Message De-Crypted

In addition to its other failings, the System Intrinsics Manual does not explain the exact reason for and/or work-around for most file system errors. In fact, most file system error messages are very hard to understand. The following is an attempt at an adequate explanation of the causes, effects, and work-arounds for different file system errors that pertain to disc files:

0 END OF FILE (FSERR 0): This error is encountered when a program attempts to read beyond the end of file or write beyond the file limit. WORKAROUND: Change the program or the file.

1 ILLEGAL DB REGISTER SETTING (FSERR 1): Should never occur for non-privileged mode

programs. For privileged mode programs, this means that the programmer attempted to do an FFILEINFO, FGETINFO, FOPEN, or FRENAME in split-stack mode (i.e., after a call to the EXCHANGEDB or SWITCHDB procedures). WORKAROUND: Do not perform the function in split-stack mode.

2 ILLEGAL CAPABILITY (FSERR 2): A function that requires privileged mode capability (e.g., open file for NOWAIT I/O, open file for EXECUTE access, etc.) was attempted without privileged mode capability. WORKAROUND: Enter privileged mode before executing the function or do not attempt to execute it at all.

8 ILLEGAL PARAMETER VALUE (FSERR 8): Parameters specified on the FOPEN call are mutually contradictory; for instance, an attempt to open a file NOWAIT on a serial disc was detected, or the program tried to open a new KSAM file without specifying the FORMALDESIGNATOR or KSAMPARAM parameters on the FOPEN. WORKAROUND: Correct the parameter.

9 INVALID FILE TYPE SPECIFIED IN FOPTIONS (FSERR 9): The file type field of the FOPEN file options is not one of 0 (STD = standard file), 1 (KSAM file), 2 (RIO file), 4 (CIR = cir cular file), or 6 (MSG = message file). WORKAROUND: Correct the file type field.

10 INVALID RECORD SIZE SPECIFICATION (FSERR 10): The record size requested was more than 32767 bytes. WORKAROUND: Specify a smaller record size.

11 INVALID RESULTANT BLOCK SIZE (FSERR 11): If the user request were honored, the block size (BLOCK FACTOR * RECORD SIZE) of the resultant file would be greater than 32767 bytes. WORKAROUND: Specify a smaller record size or block factor.

12 RECORD NUMBER OUT OF RANGE (FSERR 12): The user passed a negative record number to the FPOINT, FREADDIR, or FWRITEDIR intrinsic — this is illegal. WORKAROUND: Correct your program.

22 SOFTWARE TIME-OUT (FSERR 22): The user tried to read an empty message file or write to a full message file, an action which would cause the user to be impeded until the file stopped being empty or full, respectively (see MPE IV INTRINSICS MANUAL). However, a time out was set with the FCONTROL intrinsic (mode 4) and the request timed out before it could be honored. WORKAROUND: Do not set the time out or ensure that the request can be serviced before it times out.

26 TRANSMISSION ERROR (FSERR 26): Hardware failure. WORKAROUND: Call your CE.

30 UNIT FAILURE (FSERR 30): Hardware failure. WORKAROUND: Call your CE.

40 OPERATION INCONSISTENT WITH ACCESS TYPE (FSERR 40): The access type specified at FOPEN time does not permit this operation; fori nstance, an FWRITE is not permitted when a file is opened with ACC=IN. WORKAROUND: Specify an access type at FOPEN time that permits this operation or do not perform the operation at all.

41 OPERATION INCONSISTENT WITH RECORD TYPE (FSERR 41): It seems that this error should never show up and is merely a left-over from a previous version of MPE.

42 OPERATION INCONSISTENT WITH DEVICE TYPE (FSERR 42): The program tried to execute an operation that is incompatible with the device that it is trying to perform it on; for instance, it is trying to read the line printer or change the baud rate of a disc drive. WORKAROUND: Do not execute the operation.

43 WRITE EXCEEDS RECORD SIZE (FSERR 43): An attempt was made to write a record that would not fit in the destination file, e.g., to write a 100-byte record into a file of record length of 80 bytes. WORKAROUND: Change the file's record size, change the length of the record to be written, or open the file with the Multi-Record (MR) access option.

44 UPDATE AT RECORD ZERO (FSERR 44): The FUPDATE intrinsic (which is equivalent to the COBOL REWRITE statement) was called with the record pointer at record 0, which indicates that no record has been read and thus no record can be updated. WORKAROUND: Call FPOINT or FREAD before the FUPDATE call.

45 PRIVILEGED FILE VIOLATION (FSERR 45): A program attempted to open a privileged file (one with a negative file code; e.g., an IMAGE file) while specifying a filecode not equal to the file's filecode or while not in privileged mode. WORKAROUND: Enter privileged mode before the call or specify the correct filecode.

46 OUT OF DISC SPACE (FSERR 46): The device class on which this file resides (if this error is gotten at extent allocation time) or is requested to reside (if this error is gotten at file creation time) does not have enough contiguous disc space to accommodate this file; i.e., if NUMEXTS is the number of extents to be allocated and EXTSIZE is the size (in sectors) of one extent, this device class does not have NUMEXTS contiguous chunks of EXTSIZE sectors each. WORKAROUND: Move the file to another, less full, device class, decrease the requested file size, or decrease the extent size by increasing the number of extents in the file.

47 I/O ERROR ON FILE LABEL (FSER R 47): The internal file label of this file can not be accessed. Most likely, the file is totally cloberred and will return INVALID FILE LABEL (FSERR 108) when it is subsequently accessed. WORKAROUND: None.

48 OPERATION INVALID DUE TO MULTIPLE FILE ACCESS (FSERR 48): One of the following conditions is true: 1) The program is trying to purge (i.e., close with disposition DEL) a file that is currently loaded or being stored/restored, 2) The program is trying to rename (with the FRENAME intrinsic) a file that it does not have exclusive access to, or 3) The program is trying to open with LOCK access a file that someone else has opened with NOLOCK access or vice versa. WORKAROUND: 1) Don't purge the file or wait for the file to become purgeable again, 2) Don't rename the file or open the file with EXC access, or 3) Open the file with LOCK or NOLOCK access (whichever is the one with which the other program has the file open).

49 UNIMPLEMENTED FUNCTION (FSERR 49): The program specified an invalid parameter value in a file system intrinsic call; e.g., a disposition of 5, 6, to 7 at FCLOSE time or a file type of RIO on pre-Athena systems (ones which do not support RIO files). WORKAROUND: Correct your program.

50 NONEXISTENT ACCOUNT (FSERR 50): An attempt was made to open a file in an account which was not configured in the system. WORKAROUND: Correct the filename or build the account.

51 NONEXISTENT GROUP (FSERR 51): An attempt was made to open a file in a group which was not configured in the system. WORKAROUND: Correct the filename or build the group.

52 NONEXISTENT PERMANENT FILE (FSERR 52): An attempt was made to open a file which does not exist. WORKAROUND: Correct the program or build the file.

53 NONEXISTENT TEMPORARY FILE (FSERR 53): The program tried to open a temporary file which does not exist. WORKAROUND: Correct the program or build the file.

54 INVALID FILE REFERENCE (FSERR 54): The program tried to open a file whos e filename was invalid; for instance, the file, group, or account name was longer than 8 characters long, an invalid system-defined file was specified (e.g., $XYZZY), or no file equation was found for a back-refenced file (e.g., *MANSION with no file equation for file MANSION). WORKAROUND: Correct the filename specified. NEED

56 INVALID DEVICE SPECIFICATION (FSERR 56): The device number or device class on which the file was to be opened is not configured on the system. WORKAROUND: Correct the program.

57 OUT OF VIRTUAL MEMORY (FSERR 57): The buffer size (NUMBER OF BUFFERS * RECORD SIZE * BLOCKING FACTOR) of the file to be opened exceeds 8,192 words (or 14,000 words starting with the D MIT version of MPE). WORKAROUND: Decrease number of buffers (by specifying BUF=1 on a :FILE equation, for instance), decrease the record size of the file, or decrease the blocking factor of the file.

58 NO PASSED FILE (FSE RR 58): The program attempted to open $OLDPASS, but no $OLDPASS file exists. WORKAROUND: Correct the program or build a $OLDPASS file.

60 GLOBAL RIN UNAVAILABLE (FSERR 60): The program requested dynamic locking at file open time, but the RIN (Resource Identification Number) necessary for dynamic locking could not be gotten. WORKAROUND: Free some global RINs (with the :FREERIN command), file RINs (by closing files opened with LOCK access), open the file with NOLOCK access, or enlarge the RIN table.

61 OUT OF GROUP DISC SPACE (FSERR 61): The program tried to allocate more disc space than is allowed for a given group; e.g., it tried to build a 10,000-sector file in a group which already had 95,000 sectors and was limited to 100,000 sectors. WORKAROUND: Decrease the amount of disc space used by files in that group (by purging or squeezing files) or a ask the account manager to increase the group disc space limit.

62 OUT OF ACCOUNT DISC SPACE (FSERR 62): The program tried to allocate more disc space than is permitted for the account in which it tried to allocate it. WORKAROUND: Decrease the amount of disc space used by files in that account (by purging or squeezing files) or ask the system manager to increase the account disc space limit.

64 USER LACKS MULTI-RIN CAPABILITY (FSERR 64): The program was not :PREPed with MR (Multi-Rin) capability, yet tried to lock a file when another file (or RIN) was already locked by that program. WORKAROUND: :PREP the program with MR capability or do not try to lock a file when you have already locked another one.

71 TOO MANY FILES OPEN (FSERR 71): The program attempted to open a file, but there was not enough room in the system area (PCBX) of the program's stack for the information for that file. WORKAROUND: Close some no longer necessary files before trying the open, or run the program with the ;NOCB keyword on the :RUN.

72 INVALID FILE NUMBER (FSERR 72): An attempt was made to access (e.g., read or write) a

file that has not been opened or that is a privileged file; for instance, a read was requested against file number 10, but no file is opened as file number 10. WORKAROUND: Correct your program or enter privileged mode before trying to access the file (if the file is privileged).

73 BOUNDS VIOLATION (FSERR 73): You are attempting to read or write more data than could fit into your I/O buffer (e.g., you are trying to read 100 words into an 80-word array). WORKAROUND: Decrease the length of the data you are trying to read or write or enlarge your program's I/O buffer.

74 NO ROOM LEFT IN STACK SEGMENT FOR ANOTHER FILE ENTRY (FSERR 74): See file system error number 71 above.

90 EXCLUSIVE VIOLATION: FILE BEING ACCESSED

(FSERR 90): Exclusive access was requested to a file which is already being accessed; thus, exclusive access cannot be granted. WORKAROUND: Specify SHR (shared) or EAR (exclusive — allow read) access when opening the file or wait for the accessor to close the file.

91 EXCLUSIVE VIOLATION: FILE BEING ACCESSED EXCLUSIVELY (FSERR 91): Access was requested to a file which is being accessed exclusively by some other user. WORKAROUND: Wait for the accessor to close the file.

92 LOCKWORD VIOLATION (FSERR 92): An invalid lockword was specified at file open time or when the file system prompted the user for a lockword. WORKAROUND: Specify a correct lockword or remove or change the lockword on the disc file.

93 SECURITY VIOLATION (FSERR 93): Permitting the user to access this file in the specified access mode would be a breach of file security. WORKAROUND: Change the access mode specified in the program to one which is permitted or ask the file's creator to :RELEASE or :ALTSEC the file.

94 USER IS NOT CREATOR (FSERR 94): An attempt was made to :RENAME or FRENAME a file by someone other than the file's creator. WORKAROUND: Do not perform the :RENAME or FRENAME, ask the creator of the file to do the :RENAME, or (if you have read and write access to the file and are a user of MPEX/3000) use MPEX's %RENAME command.

96 DISC I/O ERROR (FSERR 96): Hardware failure. WORKAROUND: None.

100 DUPLICATE PERMANENT FILE NAME (FSERR 100): The program tried to save (close with SAVE disposition) a new or temporary file as a permanent file, but a permanent file with that name already exists. WORKAROUND: Purge the other file with that name.

101 DUPLICATE TEMPORARY FILE NAME (FSERR 101): The program tried to save as temporary file (close with TEMP disposition) a new file, but a temporary file with that name already exists. WORKAROUND: Purge the other temporary file with that name.

102 DIRECTORY I/O ERROR (FSERR 102): The directory (or part of it is cloberred. You're in big trouble. WORKAROUND: None.

103 PERMANENT FILE DIRECTORY OVERFLOW (FSERR 103): There is no more room in the system file directory for this file (the system file directory typically allows approximately 1200 files per group). WORKAROUND: Purge some of the files in the group in which you wish to build the file.

104 TEMPORARY FILE DIRECTORY OVERFLOW (FSERR 104): There is no more room in your job/session temporary file directory for this file. WORKAROUND: Purge some temporary files or :RESET some :FILE equations or :CRESET some :CLINE equations.

105 BAD VARIABLE BLOCK STRUCTURE (FSERR 105): The variable record length file being accessed has an inconsistent structure or would have an inconsistent structure if this access were to go through (if you are writing NOBUF). WORKAROUND: If you are writing NOBUF, correct your program; otherwise, none.

106 EXTENT SIZE EXCEEDS MAXIMUM (FSERR 106): The program attempted to build a file which would have extents larger than 65534 sectors, the maximum permitted. WORKAROUND: Increase the number of extents in the file or decrease the extent size by decreasing the record size or file limit of the file.

107 INSUFFICIENT SPACE FOR USER LABELS (FSERR 107): The maximum number of user labels for a file is 254. WORKAROUND: Decrease the number of user labels requested by the program.

108 INVALID FILE LABEL (FSERR 108): The file is inaccessible because the file is invalid (probably irrecoverably destroyed). WORKAROUND: None.

109 INVALID CARRIAGE CONTROL (FSERR 109): The program tried to do a write with a CCTL code of 1 (imbedded CCTL) but with a buffer length of 0; or, the program attempted an FCONTROL mode 1 (transfer CCTL code) with a parameter of 1. WORKAROUND: Correct the program.

110 ATTEMPT TO SAVE PERMANENT FILE AS TEMPORARY (FSERR 110): An attempt was made to close a permanent file with temporary

(TEMP) disposition; this is illegal. WORKAROUND: Correct the program.

148 INACTIVE RIO RECORD (FSERR 148): An FPOINT, FREADDIR, or FSPACE positioned the record pointer at an inactive record in an RIO (Relative I/O) file. WORKAROUND: None necessary.

149 MISSING ITEM NUMBER OR RETURN-VARIABLE (FSERR 148): An item number was specified without a corresponding variable or vice versa in an FFILEINFO intrinsic call. WORKAROUND: Correct the program.

150 INVALID ITEM NUMBER (FSERR 15 0): An item number specified in an FFILEINFO intrinsic call is invalid. WORKAROUND: Correct the program.

151 CURRENT RECORD WAS THE LAST RECORD WRITTEN BEFORE THE SYSTEM CRASHED (FSERR 151): The current record in the MSG (message) file was the last one written before the system crashed and may contain invalid information.

ACKNOWLEDGEMENT

Praises and kudos GOTO: Robert Saunders (of the HP lab) for much important information; Vladimir Volokh (of VESOFT Consultants), Robert Green (of ROBELLE Consulting), and many others for comments, questions, criticisms, suggestions, and overall moral support.