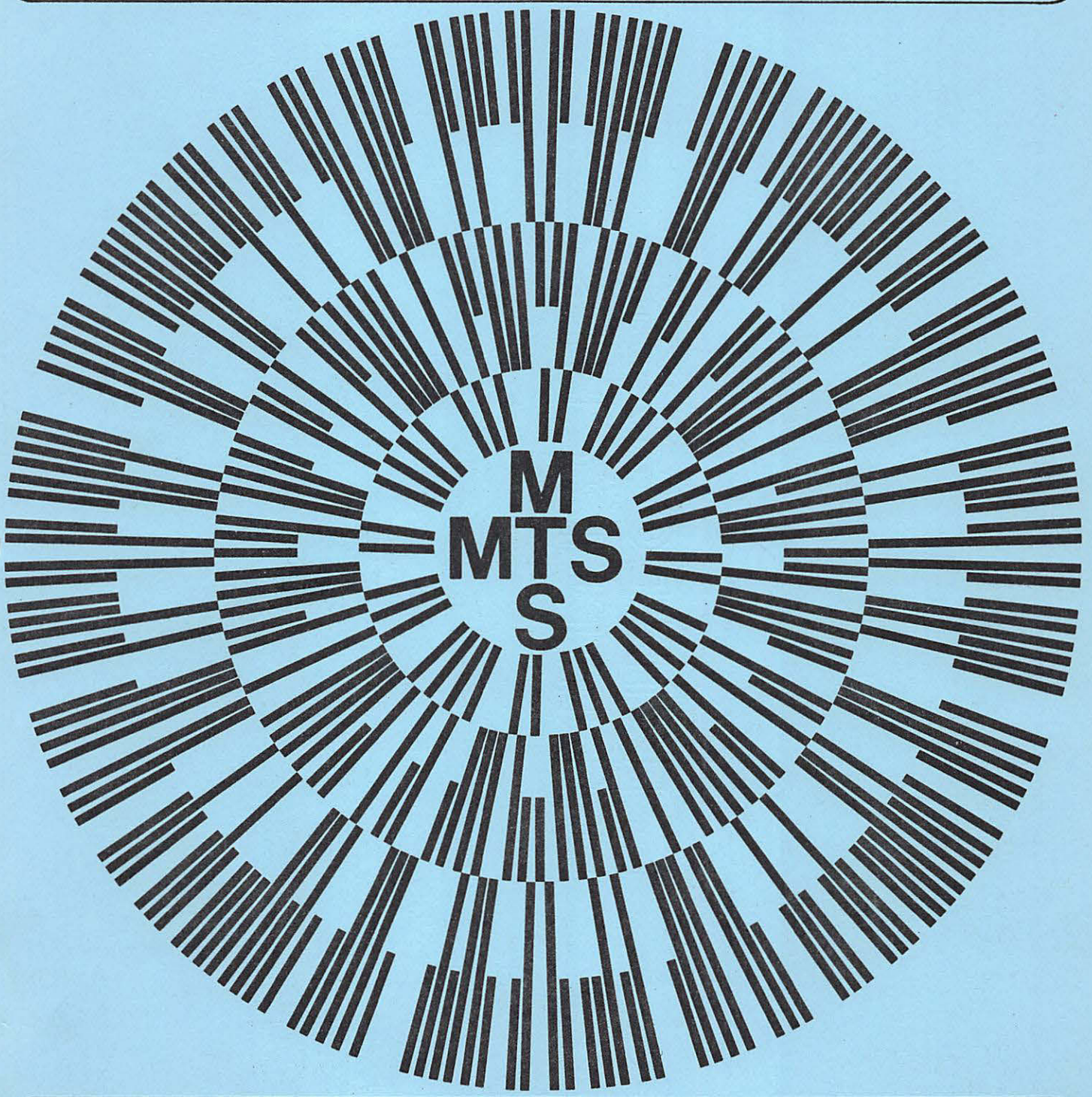
THE UNIVERSITY OF ALBERTA
COMPUTING CENTER
PUBLICATION



FILES AND DEVICES

ACKNOWLEDGEMENTS

THIS MANUAL WAS LARGELY COMPILED FROM MATERIAL PREPARED BY THE STAFF OF THE UNIVERSITY OF MICHIGAN COMPUTING CENTER. THEIR DOCUMENTATION WAS INVALUABLE AND WE ARE INDEBTED TO THEM FOR ALLOWING US TO USE IT. IN PARTICULAR, THE FOLLOWING WERE MOST USEFUL:

MTS USERS' MANUAL, SECOND EDITION, VOLUMES I AND II

MTS USERS' MANUAL, THIRD EDITION, VOLUME 2

INTRODUCTION TO MTS AND THE COMPUTING CENTER (FLANIGAN)

COMPUTING CENTER NEWS ITEMS

COMPUTING CENTER MEMOS

THE COMPUTING CENTER WISHES TO PERSONALLY ACKNOWLEDGE THE ASSISTANCE OF MIKE ALEXANDER AND DON BOETTNER WHO HELPED US TO ESTABLISH MTS AT THE UNIVERSITY OF ALBERTA.

ACKNOWLEDGEMENT SHOULD ALSO BE MADE TO THE COMPUTING CENTRE, UNIVERSITY OF BRITISH COLUMBIA, FOR INFORMATION OBTAINED FROM SOME OF THEIR DOCUMENTATION AND TO I.B.M., WHOSE MANUALS PROVIDED CERTAIN SECTIONS FOR OUR MANUALS.

FILES AND DEVICES

MAY, 1970

DISCLAIMER

This MTS manual is a combination of earlier manuals, update notices, memos and limited experience with the system itself. Because of this, certain discrepancies are bound to occur and the Computing Center would appreciate being notified of all differences between what this manual says and what the system actually does.

This publication is intended to represent the current state-of-the-system. However, it should not be construed as an obligation to maintain the system as so stated. The MTS system, like most good systems, is continually being improved. As a result, additions, extentions, changes and deletions will occur. Notice of such changes will be made and provision for a manual updating service has been planned.

Errors, comments and suggestions should be sent to:

Information Coordinator
Computing Center
University of Alberta

FILES AND DEVICES
MAY, 1970

TABLE OF CONTENTS

1. Introduction 1.1

2. Concepts, Definitions and Conventions 2.1
   - Files and file names 2.1
   - Device names 2.1
   - Logical I/O units 2.2
   - Pseudo-device names 2.3
   - Logical I/O unit defaults 2.5
   - Representation of names 2.5
   - Interpretation of names 2.6
   - Modifiers 2.6
   - Implicit concatenation 2.8
   - Explicit concatenation 2.8
   - Summary of names and naming conventions 2.9

3. File Organization 3.1
   - Line files 3.2
   - Sequential files 3.4

4. Lines 4.1
   - Command lines 4.1
   - Data lines 4.1
   - Line numbers 4.1
   - Continuation lines 4.2
   - Line trimming 4.3

5. File Manipulation 5.1
   - Creating files 5.1
   - Emptying and destroying files 5.2
   - Listing and copying files 5.3
   - Placing information into files 5.10
   - Placing information into a currently active line file 5.10
   - Automatic numbering of data lines 5.12
   - Putting data into a currently active sequential file 5.15
   - Placing MTS commands in a file 5.16

6. File Routines 6.1
   - Sequential files 6.1

7. Internal File Structures 7.1

8. Utilities

*CATALOG 8.1
*FILEDUMP 8.3
*FILESCAN 8.5
*FILESNIFF 8.7
*FILEUSE 8.9
*FSAVE 8.11
*PERMIT 8.15

1. INTRODUCTION

This manual should provide the user with all the necessary conceptual material he may require when using files or devices at the command level. It does not attempt to provide complete information on file manipulation at the command or subroutine levels. The user is referred to the COMMANDS and SUBROUTINE LIBRARIES manuals for details.

2. CONCEPTS, DEFINITIONS AND CONVENTIONS

Files and File Names

Programs and data in MTS are stored in files, where a file is an ordered set of zero or more lines. A line is a string of 1 to 255 characters or bytes (for line files) or a string of 1 to 32767 characters or bytes (for sequential files). The distinction between line files and sequential files will be discussed in a later section of this manual; for the present purposes, it is unnecessary to make this distinction. All long-term storage in MTS is organized on the basis of files and hence is referred to as file storage; this applies to source decks, object decks, data sets, and/or what have you. In MTS there are "public" and "private" files; private files are those created by a user for storage of his private decks. Private files may be temporary, lasting only during part of a single job, or permanent, remaining in the system for as long as several months. Permanent private files are created and named at the command of the user; temporary files are created whenever a temporary file name is encountered by MTS in the user's deck. Temporary files are created by the mention of their name and destroyed when the job in which they were created is terminated. Public files contain the components of the MTS system. Public files are also known as "system" files, and private files are sometimes called "user" files.

Permanent private file names consist of from one to twelve characters. If more than twelve characters are given, the first twelve are used. The name cannot contain blanks, commas, or left parentheses, plus signs, or semicolons, and any lower-case letters are automatically translated to their upper-case equivalent. It is strongly recommended that names assigned be alphanumeric only, i.e., no special characters, since the specification may change in the future.

Temporary or scratch file names consist of from two to nine characters, the first of which is a minus sign (-). They are automatically created the first time they are mentioned and are destroyed when a user is signed off.

Public file names consist of an asterisk followed by from one to fifteen characters, the last of which is not an asterisk. Library files may be accessed by users but are protected against modification.

Device Names

Device names are system symbols which name specific hardware devices which are attached to the computer; each I/O device on the 360/67 has a distinct name. To specify the location of a program or data set, the user may specify the name of the file in which the program or data set is stored, or he may specify the name of the device which is ready to input the program or data set.

Device names are four characters long (padded with trailing blanks, if necessary). In general, pseudo-device names (see next section) are to be used instead of device names when physical devices must be referred to. Some of the device names currently recognized are:

Device Names	Device Types
LA30 through LA87	2703
OPER	1052 console
PCH1	card punch
PCH2	card punch
PTR1	line printer
PTR2	line printer
RDR1	card reader
RDR2	card reader
TOC0	7 track tape drive
TOC1	7 track tape drive
TOC2 through TOC7	9 track tape drives

Logical I/O Units

When one writes a program, the file or device name for the program's data may be unknown, so that it becomes impossible to supply such information at translation time. Even if this information were known, however, it would be inconvenient to specify it during translation, since this would require re-translation every time the file or device is changed. Thus, it is desirable to specify the location of data at execution time rather than at translation time. To do this, we introduce the concept of a logical I/O unit: a logical I/O unit is a symbolic name which is used in a program to specify the source of data for input or the destination of output information. A logical I/O unit does not name a specific file or device; it simply serves as a reference. When a program is run, it becomes necessary to first specify, for each logical I/O unit used by the program, which actual file or device should be used. For example, in FORTRAN IV the statement

```
READ (5,100)P
```

requests input from logical I/O unit 5. Before this statement can be executed, the user must specify a file name or device name to be used whenever logical I/O unit 5 is referenced. (Note: in the FORTRAN IV language manual, the 5 in the READ statement is called a "data set reference number". This number appears in MTS as a logical I/O unit, however, and this latter terminology will be used throughout this manual.

The names which may be used as MTS logical I/O units are:

the integers 0 through 9
SCARDS
SPRINT
SPUNCH
SERCOM
GUSER

The various translators normally use some or all of these logical I/O units. User programs written in Assembler language can use any of the logical I/O units.

Pseudo-device Names

In MTS, execution of a program normally is initiated by a $RUN command; this command requests execution of the program and provides MTS with the name of the file or device where the program can be found. This procedure is followed both for system programs (such as translators) and for user programs, The $RUN command must specify not only the object program to be run, but also the file and/or device names to be used for the references to the various logical I/O units used by the program. Thus, to run the FORTRAN IV translator the $RUN command must specify where the source deck is to be found and where the object deck is to be placed, along with other information.

This naming requirement in the $RUN command creates immediate problems for the batch user in MTS. Since in a batch run the input batch deck has been read by HASP and placed in an unknown (to the user) file before it is executed, the batch user has no way of providing in the $RUN command a specific file name for the file which contains his source deck and/or data cards. This problem is circumvented by pseudo-device names.

Pseudo-device names are synonymns for true files or devices. They consist of an asterisk followed by from two to fourteen characters, the last of which is an asterisk. Seven of these names described below are automatically defined for a user and additional ones may be defined using the public file *MOUNT (see the Tape Users' Guide). The predefined pseudo-device names are:

SOURCE is defined by MTS as the current input source file or device. Initially, the system defines *SOURCE* to be the same as *MSOURCE*. For batch runs *SOURCE* is defined as the appoopriate HASP input file which contains the user's input deck, while for conversational runs *SOURCE* is defined as the terminal at which the user is signed on. Therefore, any reference by a batch user to *SOURCE* is equivalent to a reference to the HASP input file containing the appropriate batch input deck. The batch user may thus use the name *SOURCE* in place of the name of the HASP input file which he doesn't know.

The user can redefine *SOURCE* using the $SOURCE command. If *SOURCE* has been redefined by a user, an attention interupt at a terminal or an end of file on *SOURCE* when attempting to read a command will cause *SOURCE* to be redefined the same as *MSOURCE*.

SINK is defined as the current output (or sink) file. Initially, the system defines *SINK* to be the same as *MSINK*. For a batch job, MTS defines *SINK* as the appropriate HASP OUTPUT file for that job, while for conversational jobs *SINK* is defined as the user's terminal. Thus, the user need not know the specific name of the HASP output file being used to collect his batch output, since he may refer to it using the pseudo-device nmme *SINK*.

The user can redefine *SINK* using the $SINK command. If *SINK* has been redfined by a user at a terminal, an attention interrupt will cause *SINK* to be redefined the same as *MSINK*.

PUNCH is defined as the current HASP output punch file for a batch run. MTS does not define *PUNCH* for conversational runs and any reference to it will be treated like an undefined file. The batch user may use *PUNCH* to place card output in the appropriate HASP punch output file for his batch job.

AFD is defined as the current active file or device (that file or device which was obtained by $GET or $CREATE), if there is one.

DUMMY is defined as an infinite wastebasket for output (lines are accepted and they disappear) and an empty data set for input (every time a line is requested, an end-of-file condition is returned). *DUMMY* is particularly convenient for specifying at run time that run output on some logical I/O unit is to be ignored.

MSINK is defined as the master output (or sink) file or device which is the terminal for conversational operation and the printer for batch operation.

MSOURCE is defined as the master input (or source) file or device, which is the terminal in conversational operation and the file or device from which the batch job is being read in batch operation.

The use of these pseudo-device names solves our earlier problem; one may now run the FORTRAN IV complier and inform it that the source deck is on *SOURCE* and the output from the compilation is to be written on *SINK*. These pseudo-device names may also be used to indicate the input data source and the output result destination during execution.

The use of the three pseudo-device names *SOURCE*, *SINK*, and *PUNCH* allows the batch user to properly address the appropriate input and output files being used for his batch job even though HASP itself is setting up and maintaining these files.

Logical I/O Unit Defaults

Since it is convenient to reduce the amount of keypunching necessary to specify the various pieces of information required on a $RUN command, some of the logical I/O units have default specifications; these are assignments of pseudo-device names to logical I/O units if no other assignments are given on the $RUN command.

Logical I/O Unit	Default Specification
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH* (defined for batch only)

For example, if on a $RUN command the logical I/O unit SCARDS is not specified as some particular file or device, then it is by default assigned to the current input file *SOURCE*. Since most of the translators available in MTS use SCARDS for source program input and SPRINT for compilation listing output, it is often unnecessary for batch users to specify these logical I/O units when running the translators. Terminal users need only specify these logical I/O units when they are to be other than the terminal itself. Both in batch and in terminal use, it is usually necessary to specify the logical I/O unit for translator output of the resulting object decks. Logical I/O units 0 through 9 have no default specifications in MTS itself; there are, however, default specifications for these logical I/O units within the FORTRAN IV I/O routines during program execution (see the FORTRAN Users' Guide).

Representation of Names

File names and device names (real or pseudo) are interchangeable. That is, a file name may in general be used anywhere a device name may appear, and the converse. The generic quantity

<FDname>

used in these manuals will signify such an interchangeable designation or which may be a concatenation of file and device names, and

<Fname> or <Dname> or <PDname>

will be used if cases arise where a restriction to a single (i.e., no concatenation) file, real device, or pseudo-device name (respectively) is necessary.

Interpretation of Names

All FDnames entered in commands, etc. will be interpreted as file names, even if they happen to be identical to device names, thus avoiding confusion between a file and a device with the same name.

If it is desired to specify that a name is to be taken only as a device, the name should be prefixed with a >. For example,

RDR1 refers to file RDR1
>RDR1 refers to reader 1

If it is desired to specify that a name is to be taken as a file only, the name should be prefixed with a #. In the example above,

#RDR1 refers to file RDR1

Modifiers

The action of a file or a device may be changed by appending to the file or device name one or more modifiers. Each modifier consists of an at-sign ("@") [mnemonic for "attribute"], followed by the modifier name as given in the following table. A modifier name may be preceded by a not-sign ("¬") to reverse its meaning.

If any modifier is given explicitly in a call to an I/O subroutine the setting given in the call overrides the setting given with the FDname.

If an illegal modifier is given for a file or device name, the user will be prompted to replace the file or device name (if he is on a terminal) when it is first used. A batch user will not be allowed to use the file or device name with the illegal modifier.

In the following table, the underlined portion of the modifier name is the abbreviation.

Name	Explanation of modifiers
SEQUENTIAL INDEXED	Indexed and Sequential are opposites. Indexed means the user is specifying the line number; i.e., he says "put (or get) this line at (from) this place". Sequential means that the user wants to get (or put) the next line.
EBCD BINARY	Normal mode is EBCD. The binary modifier is used to read or punch binary cards. It has nothing to do with binary tape. Binary can not be used by a batch job run under HASP which reads cards from the input deck i.e., binary modifier applied to *SOURCE* or *MSOURCE*.
LC UC	Normally characters are transmitted unchanged. UC means lower case letters are changed to upper case letters.

CC — Controls carriage control for printer. Normal carriage control is "logical carriage control". Legal carriage controls are:

+	overprint (i.e. no space)
blank	single space
-	triple space
0	double space
9	single space and suppress overflow
1	skip to next page
2	skip to next half page
4	skip to next quarter page
6,8	skip to next sixth page
&	print line without carriage return (for all terminals)

SS — Controls stacker select for punch. If stacker select is specified, the first character is taken as control;

0	stacker 1
1	stacker 2
2	stacker 3

If first character is a legal control it is so used, and characters 2 and on are punched in column 1 and on. If it is not a legal control, the card is punched as is (character 1 and on into column 1 and on) and stacked in stacker 2.

CC — Control function for magnetic tape drives (see Tape Users' Guide)

PREFIX or PFX — If on, for both input and output, the current line number is converted to external form and printed as a prefix on the appropriate input or output line.

PEEL — If on, for input, a line number is "peeled" off the front of the line, converted to internal form and returned. The rest of the line is returned as the input line.
If on, for output, it forces the line number in the sequential case to be returned to the spot designated in the parameter list. (not normally the case).

MCC — If on, then machine carriage control is used. In this case, for printers (and simulated for terminals), the first byte of the line is used as the command in the CCW used for output, if legal, and the first byte is blanked out. If not legal, the entire line is printed single spaced. For all other devices and for files, this modifier is ignored. The following machine carriage control characters for skip to channel 9 or 12 are illegal:
x'CB', x'E3', x'C9', x'E1'

TRIM	Normally TRIM is on. This means that if a line has any trailing blanks, all but one are deleted.
SPECIAL	This modifier is reserved for device dependent uses. Its meaning depends on the particular device type it is used with. If applied to an output operation on a sequential file, the line will replace the one at the current write pointer instead of causing the file to be truncated at this point. The replacement line must be the same length as the original line or an error comment (which can be intercepted with @ERRRTN or SETIOERR) will be produced. This feature is most useful in conjunction with the subroutines NOTE and POINT.
IC	If on, implicit concatenation occurs during I/O operations. If off, implicit concatenation does not occur during I/O operations. The modifier overrides, but does not reset, the implicit concatenation switch ($SET IC=ON or $SET IC=OFF) when applied to FDnames or I/O calls.

Implicit Contenation

Although the maximum size of any single line file is limited to approximately 7000 to 12000 lines, the amount of data that may referred to by a given file name is effectively unlimited because several files may be automatically chained together in two ways. Whenever a line beginning with the character string

```
$CONTINUE WITH FDname
```

is read from any file or device, reading continues with the file or device named and the $CONTINUE WITH line is not passed to the caller. (See the $SET command for a way to override this action). This is called implicit contenation.

If the FDname on the $CONTINUE WITH line is followed by the word RETURN (separated from FDname by a blank) then the lines in the file "continued with" are substituted for the $CONTINUE WITH and then reading continues with the line following the $CONTINUE WITH line. Without the RETURN, data following the $CONTINUE WITH line is ignored.

Explicit Concatenation

Several files and/or devices may be chained together by using explicit concatenation. This is done by giving the names of the files and devices with their modifiers (and line number ranges for line files) in the order desired separated by "plus" signs. For example

```
-LOAD(1,100)+*SSPLIB
```

means the contents of lines 1 through 100 of -LOAD followed by the contents of *SSPLIB.

If two or more consecutive line file names in an explicit concatenation are the same, all but the first may be omitted. For example

```
A(1,1)+(LAST)
```

is the same as

```
A(1,1)+A(LAST)
```

If one or more members of an explicit concatenation uses implicit concatenation (i.e. contains $CONTINUE WITH FDname) the entire implicit concatenation is used as that portion of the explicit concatenation. In this case the FDname in the $CONTINUE WITH may itself be an explicit concatenation. The processing of the next member of an explicit concatenation is started whenever a return code of 4 (end-of-file) is received on a read operation or the ending line number of the current member is exceeded on a sequential write operation. Care should be exercised when using indexed operations on a concatenated file.

If some member of an explicit concatenation does not exist or is not available and the user is not running in batch, he will be given a chance to enter a replacement name the first time this member is used. This replacement name may be any explicit concatenation of files or devices and replaces only the one member of the original concatenation.

Some examples of explicit concatenation are:

```
MAIN+SUBR
DATA(1,1)+(3,10)@UC
*SOURCE*+DATA+ALLOC(1,10)
*TAPE*(1,10000)+*TAPE2*
```

It should be noted that explicit concatenation of file names is effectively a new file name of short duration. That is, the explicit concatenation

```
MYSOURCE+-DATA+*SOURCE*
```

is effectively the name of a file consisting of the contents of the permanent file MYSOURCE, followed by the contents of the temporary file -DATA, followed by the information on the pseudo-device *SOURCE*.

Summary of Names and Naming Conventions

In summary of the last several paragraphs, MTS contains the following entities for naming files and/or devices:

file name	name of a specific file which someone has created and named; may be a system file or a user file;
device name	name of a specific hardware I/O device on the 360/67; these names are defined in MTS and cannot be changed by the user;

logical I/O unit	symbol which may be used in a program like a file or device name, but which does not represent any specific file or device until the user specifies a unique file or device name to be used for all references to the pertinent logical I/O unit;
pseudo-device name	symbol which may be used in an MTS command like a file or device name, but whose assignment to a specific file or device is controlled by MTS and depends upon the current status of the processing and the commands which have been executed.

To distinguish between these various file types, MTS imposes a naming convention on files which allows the file types to be determined from the file name. The MTS naming convention is as follows:

1. User permanent files: file name must consist of 1 through 12 alphanumeric characters; lower-case letters are converted to upper-case; some special characters are allowed in user permanent file names, but this should be avoided by the user; in the system file directory, the user's CCID is prefixed to the user permanent file name to provide unique names among the many MTS users; example user permanent file names are MYFLLE, FORTRANSOURCT12, and LIBONE;
2. User temporary files: file name must consist of 2 through 9 characters, the first of which is a minus sign (-), and the remainder of which are alphanumeric characters; the leading minus sign identifies a user temporary file name; example user temporary file names are -T, -TEMPORAY, -LOAD#, and -OBJECT;
3. System files: file name consists of an asterisk (*) followed by 1 to 15 characters, the last of which is not an asterisk; users cannot create, modify, or destroy system files; a system file name isrecognized by the leading asterisk with no trailing asterisk; example system file names are *SWAT, *ASMG, *EDIT, and *LIBRARY;
4. Pseudo-device names: name consists of an asterisk (*) followed by 2 through 14 characters, the last of which is an asterisk; the user may create such names at times (such as in using tape files), and there are a number of pseudo-device names predefined in MTS; a pseudo-device name is recognized by the leading and trailing asterisks in the name; example pseudo-device names are *SOURCE*, *SINK*, *PUNCH*, *AFD*, and *DUMMY*.

3. FILE ORGANIZATIONS

There are two types of file organization for a user's private files: line files and sequential files.

The line file organization can be characterized by saying that all lines of information have a line number associated with them and such lines must be less than 256 bytes in length. In addition, numerous editing operations are allowed on line files, such as replacing, inserting or deleting lines. Finally, it is possible to refer to subsets of a line file. All this is accomplished by either giving the appropriate line numbers to MTS via commands (e.g., $LIST *AFD*(5,7), $COPY, $GET), by attaching line numbers to I/O files (e.g., SCARDS=FILE(3,LAST)), or by giving the line numbers to certain programs which manipulate line files (e.g., *EDIT). In general, a line file is very useful for relatively short lines of information which are changed frequently and which are accessed in an arbitrary random fashion.

Sequential files, on the other hand, have much more restricted applications. For example, inserting, deleting and replacing lines of information is not possible for a sequential file. In addition, there is no easy way to refer to a subset of a sequential file. That is to say, reading from a sequential file (e.g., "COPYING FROM") always starts from the beginning of the file and writing to a sequential file (e.g., "COPYING TO") results in the information being appended to the end of the file.

Given the above restrictions, then it is not at all obvious why anyone would prefer, or even consider, using a sequential file when a line file is so much more flexible. Of course, there are definite applications for which sequential files would be preferred and to these we now turn.

First,sequential files can initially be much larger than line files and they will extend themselves (up to 16 times) in increments of the original size if necessary and have virtually no upper size limit (within the constraints of available file space and user file space allocation). Line files, on the other hand, also extend themselves when necessary, but have an absolute maximun size of approximately 12,000 - 40 byte lines. In addition, lines of information in a sequential file may be up to 32,767 bytes in length as opposed to the restriction of 256 bytes in line files. Further sequential files need not use space to store line numbers. As well reading and writing of a sequential file is faster than for a line file, since anticipatory buffering (i.e., reading ahead) is done, in addition to the fact that the overhead involved in keeping lines in order and searching for lines according to line number is not involved. However, only line files can be modified via indexed I/O operations. Line files may be converted to sequential files, but cannot be converted back to line files with the same line numbers as before. There is also a sequential-with-line-numbers file in MTS; this is a sequential file in which a line number is stored as part of the content of each line. Here a line file may be converted to such a file and then back to a

line file with the same line numbers as before.

In general, line files are used for smaller files which are often changed (such as source programs), whereas sequential files are used for large files (such as data banks) or for files which are read sequentially but seldom modified (such as object modules).

Line Files

A line file is composed of zero or more lines; a line is a sequence of 1 to 255 bytes or characters. Each line in a line file has a unique line number; a line file is ordered by the line numbers of its lines. A line number has the format

snnnnn.nnn

where s is the sign (+ or -) and n is a decimal digit (0 through 9); the minimum and maximum line numbers are then -99999.999 and 99999.999, respectively. When typing a line number, leading plus signs and zeroes, trailing decimal points, and trailing zeroes after a decimal point may be omitted. Note that while the line number of each line must be stored in the file, the line number is not part of the content of the line; line numbers are used only to reference lines in the file. By using a line number, any line in the file may be directly referenced, either for reading out of the file or for writing into the file. Whenever a line file is referenced in an MTS command, it is necessary to provide the line numbers of the first and last lines to be used in the file; if no line numbers are given, then MTS assumes it should start with line number 1 and continue to the end of the file. For example, the command

```
$RUN *FORTG SCARDS=SOURCE
```

requests a FORTRAN compilation of the source program in the file SOURCE starting with line number 1 and continuing to the end of the file. Note that any lines in the file SOURCE whose line numbers are less than 1 would not be processed. To indicate a starting line number other than 1. the line number may be placed in parenthesis after the file name. In the command

```
$RUN *FORTG SCARDS=SOURCE(-5)
```

FORTRAN will read all the lines in SOURCE starting with line number -5. One may also indicate the ending line number of a file by placing it after the beginning line number, separated by a comma. The command

```
$RUN *SWAT SCARDS=PROG(10,50)
```

causes WATFOR to compile the FORTRAN source program in lines 10 through 50 (inclusive) of file PROG. By using explicit concatenation of files with line numbers, it is possible to pick out pieces of several files. In the command

```
$RUN *ASMG SCARDS=A+B(10,20)+C(7)
```

the assembler will receive a source program composed of the contents of file A, followed by lines 10 through 20 of file B, followed by the contents of file C from line 7 to the end of C. In the command

```
$RUN *WATFOR SCARDS=A(1,10)+A(50,100)
```

WATFOR compiles the program composed of lines 1 through 10 and lines 50 through 100 of file A. A shorthand notation is available here; when an explicit file concatenation involves the same file only the new line numbers need be given. Thus, the above command may be written as

```
$RUN *WATFOR SCARDS=A(1,10)+(50,100)
```

Finally, one may also give a line number increment in a file name by placing it after the ending line number with a comma between the two. Hence, F(10,100,2) means all the even-numbered lines from 10 through 100 of file F; note that this is not necessarily every other line of F. In particular, if all lines of F have an odd line number or if all lines of F have a fractional part in their line number, then F(10,100,2) is an empty file.

In summary, anywhere that a line file name is used in MTS commands (with a few exceptions), the file name may be given in the form

```
filename(b,e,i)
```

where b and e are the beginning and ending line numbers and i is the line number increment. The defaults for b,e, and i are

```
filename(1,99999.999,1)
```

Any combination of b,e, and i may be given in a file specification. If any of the three items are omitted, trailing commas resulting from the omission of items may be left off; leading and internal commas are required.

Further, the symbol LAST may be used in b or e to indicate the last line number in the pertinent file. That is, F(LAST) is the last line of file F, while G(LAST-10) represents all lines in file G whose line numbers are greater than or equal to n-10, where n is the line number of the last line of G. In the latter case, this may be more or less than 10 lines. The term LAST+n may be used either for b or e in a file specification. In the command

```
$RUN *FORTG SCARDS=P SPUNCH=-LOAD#(LAST+1)
```

FORTRAN is directed to put the object module in file -LOAD# starting with line number LAST+1 (i.e., starting with a line number one greater than the current last line number). Hence, the object module is put at the end of -LOAD#, not disturbing its current contents. For an empty file, LAST has the value zero.

Note that all the facilities of line files discussed here may be used in implicit file concatenation, also; the line

```
$CONTINUE WITH A(10)+B(-5,0)+A(-10,9)
```

causes a continuation to lines 10 through LAST of file A, lines -5 through 0 of file B, and then lines -10 through 9 of file A.

I/O operations on a line file may be performed in one of two modes: indexed or sequential. An indexed I/O operation always involves a specific line whose line number is given as part of the I/O operation. Indexed I/O operations may thus read/write the lines of a file in any order. Note that b,e, and i values are not used in indexed I/O. In a sequential I/O operation, the lines of

the file are red/written in sequence by line number. The first line number is the one specified in the file name (b, default 1) and the last line number is the ending line number given in the file name (e, default 99999.999). Each time a new line number is needed, the line number increment given in the file name (i, default 1) is added to the last used line number to obtain the next line number to be used. In almost all cases in MTS, the standard I/O operations is sequential I/O. Note that it is the user's responsibility to ensure that the line number increment (i) used with a line file in a sequential I/O operation is sufficiently small ot reference all the desired file lines.

Sequential Files

A sequential file is composed of an ordered set of zero or more lines; a line is an ordered sequence of 1 to 32767 bytes or characters. The lines of a sequential file do not have line numbers and may not be directly referred to from a program or a command.

A sequential file may be read or written in a sequential manner only; a read always gets the next line, starting with the first line, while a write always adds the new line at the end of the sequential file.

Since no line numbers exist in a sequential file, the b,e, and i values may not be specified as part of the file name.

Further, a sequential file may not be modified other than to add lines at the end of it or to empty or destroy it.

Sequential files may be used in place of line files whenever the file is read/written sequentially. Further, sequential and line files may be interspersed in implicit and explicit file concatenation, again subject to sequential I/O only.

In essence, a sequential file behaves much like a tape file and is therefore used much like a tape.

4. LINES

Insofar as MTS is concerned, the source stream representing a job is a set of source lines composed of two types of line: command lines and data lines.

Command Lines

A command line is a line which starts with a single dollar sign ($) followed immediately by the command name (or the command abbreviation) and the command parameters. Under certain conditions in a conversational mode run only, the leading $ in a command line may be omitted (if there is no line number at the front of the input line and automatic numbering is off or there is no active file). The leading $ is required when running in batch.

When MTS recognizes a command line, it immediately performs the necessary processing to execute the command. A description of each command is given in the COMMANDS manual.

To simplify the description in this manual, it is assumed that all command lines begin with a single $. The beginning user is well-advised to start all command lines, batch or conversational, with a $ until he has gained a reasonable acquaintance with MTS.

Data Lines

Any other source line is presumed to be a data line. Also, any line which begins with two dollar signs ($$) is presumed to be a data line and is placed in the currently active file or device: in this process, one of the leading dollar signs is removed, leaving a single leading dollar sign. This allows command lines to be placed in files; examples of this will be given later. When MTS encounters a data line, it places the data line into the currently active file or device; if there is no currently active file or device, then MTS issues an error message and ignores the data line. If automatic line numbering is off, the line is inspected for a line number, and what follows the line number is put in the file as the line with that number. If there is no line number, the line is treated as a command line containing an invalid command and an error message is issued. If automatic numbering is on, the line is not inspected but is taken verbatim, except that, if the first two characters are dollar signs, only one is transmitted to the file.

Line Numbers

All lines read by the MTS monitor that are not commands must start with a line number (except when the automatic numbering mode is on). In its full form this line number contains a sign, five digits, a decimal point, and three digits.

The sign need not be specified; if missing it is assumed positive. The decimal point and following fractional digits need not be specified; if missing, the number is assumed an integer. Leading zeros need not be specified in the integer part; trailing zeros need not be specified in the fraction part. Complaint will be made if more than five digits precede the decimal point or more than three digits follow it.

Examples of line numbers are:

5 5.1 5.13 5.137 32505.137 -32505.137

The first character following the line number is the first character of the line. The end of the line number is determined as follows:

1. An alphabetic character terminates the number.
2. The second occurrence of . terminates the number (first occurrence is the decimal point).
3. A + or - which is not the first character terminates the number.
4. A blank terminates the line number.
5. Any other special character terminates the line number.

If the character which terminates the line number is a Line Number Separator character (usually a comma), then this character is considered only a separator and is not taken as either part of the line number or as part of the line. For example, to put the three characters "123" in as line one of a file,

1,123

would be typed. (See the $SET command for changing this character.)

Two other forms of line numbers are permitted. They are:

LAST
LAST±linenumber

The value of LAST is the line number of the last line in the current active file or device. Note that LAST-1 does not necessarily specify the line number of the next-to-last line; it is merely a line number 1 less than that of the last line.

Continuation Lines

To allow input lines (command or data) in the source stream which are longer than a card or a terminal line, MTS has a convention for continuation lines. If a source line terminates with a minus sign (-), then MTS deletes the minus sign and takes the next source line as a continuation of the current source line. In batch runs, the minus sign must be in column 80 of the input card; in conversational runs, the minus sign must immediately precede the end-of-line control character for the terminal being used.

Line Trimming

To reduce the number of characters transmitted, all lines going through the MTS I/O interface are trimmed of trailing blanks; that is, all trailing blanks, except one, are deleted from the end of the line. Therefore, a line will have at most one trailing blank. This applies to all lines read from the source stream and to all lines read from or written to a file.

5.0 FILE MANIPULATION

Creating Files

A file of name Fname is created in MTS by issuing the $CREATE command:

```
$CREATE Fname
```

MTS will create the file and name it Fname, making appropriate file directory entries; this command also makes the created file Fname the currently active file (see later). If for some reason the file cannot be created (e.g., user already has file of this name or user has exceeded his file space allocation), then MTS will not create the file but will print an error message stating why the file could not be created.

By default, the file Fname would be created large enough to hold about 75 40-byte lines (or the equivalent number of lines of different lengths), would be created as a line file, and would be placed on a disc. If one wishes to specify a different size for the file, he may do so with the SIZE option:

```
$CREATE Fname SIZE= n|nP|nT
```

The desired size of the file may be stated in one of three ways:

```
SIZE=n      n 40-byte lines
SIZE=nP     n 4096-byte pages
SIZE=nT     n 7294-byte tracks
```

where n is a decimal integer. For example, the command:

```
$CREATE SOURCE SIZE=500
```

creates a disc line file named SOURCE large enough to hold about 500 40-byte lines, while the command:

```
$CREATE OBJ SIZE=10P
```

creates a disc line file named OBJ of size 10 pages (40960) bytes), and the command:

```
$CREATE -TEMP SIZE=2T
```

creates a temporary disc line file named -TEMP of size 2 tracks (14588 bytes). Note the use of a temporary file name in the last command; due to the automatic creation of temporary files by MTS (as disc line files of large size), temporary files need be specifically created only when certain file attributes are to be other than default file attributes. After a file is created, MTS will enter lines into the file until the allocation file space is used. Then, MTS will attempt to allocate new space for the file, going beyond the original size estimate. If this is not possible, then an error message is issued informing the user of a file overflow. At no time

will MTS allocate for a user more permanent file space than is authorized for the user's CCID. Since there is no guarantee that MTS will be able to extend a file, the user should give the best possible SIZE estimate in creating the file.

By default, all files created are line files. To specify otherwise, the TYPE option may be used:

```
$CREATE Fname TYPE= LINE|SEQ|SEQWL
```

where the TYPE options are:

LINE	Line file (default)
SEQ	sequential file
SEQWL	sequential file with line numbers

The SIZE option must also be given if default size (70 40-byte lines) is not the desired size. Hence, the command:

```
$CREATE DATA SIZE=1T TYPE=SEQ
```

creates a disc sequential file named DATA of size 1 track, while the command:

```
$CREATE -X TYPE=SEQWL SIZE=3P
```

creates a temporary disc sequential-with-line-number file of size 3 pages.

The SIZE and TYPE options may be specified in any order and in any combination as needed by the user.

When a file is created, it is empty; that is, it contains zero lines. It is, however, good practice to never assume a file to be empty; always issue the $EMPTY command (see later) rather than making such an assumption.

Emptying and Destroying Files

Once a file has been created, it may at any time be emptied or destroyed. This applies both to permanent files and to temporary files. Permanent files exist until the user requests their destruction; temporary files exist until the end of the current run or until the user requests their destruction, whichever comes first. To empty a file of name Fname, one issues the command:

```
$EMPTY Fname
```

This command causes file Fname to be emptied, but preserves the space allocated to Fname on the disc. Future references to Fname will reuse this file space. In a conversational mode run, MTS will ask the user to confirm the $EMPTY command if Fname is a permanent user file by issuing the statement:

```
FILE "Fname " IS TO BE EMPTIED.  PLEASE CONFIRM.
```

The user may then type OK to empty the file. To cancel the command, the user may simply send back a line with no typing. In batch mode, there is no query to confirm the $EMPTY command, nor is there a query in conversational mode for temporary files. To destroy a file, one issues the command:

$DESTROY Fname

This command not only empties file Fname, but it also deallocates the file space assigned to Fname and removes Fname from the system file directory. Future references to Fname will be treated as references to an undefined file, unless Fname is again created. For conversational mode runs, there is again a need to confirm the destruction of a user permanent file. MTS will type:

FILE"Fname " IS TO BE DESTROYED. PLEASE CONFIRM.

The user responds OK to destroy the file or returns an empty line to cancel the command. Again, in batch mode no confirmation is requested, nor is confirmation requested for temporary files in conversational mode.

The $EMPTY and $DESTROY commands empty and destroy whole files; parts of files cannot be emptied or destroyed by attaching the b, e, and i values to the file name in the command. Hence, the command:

$EMPTY SOU(10,50)

is the same as the command:

$EMPTY SOU

To partially empty a file, one can copy out the desired parts of the file, empty the file, and then copy back in the desired parts (see the $COPY command described later in this section). Also, no file attributes may be changed after a file has been created. To change the TYPE, or SIZE attributes of a file, the file must be destroyed and then recreated with the new attributes.

Listing and Copying Files

There are two additional MTS commands, $LIST and $COPY, which allow the user to list or copy a file or a combination of files, either for output purposes or to create new files from old files. The $LIST command has the format:

$LIST [file1] [file2]

and causes file1 to be listed, with line numbers, on file2. If file1 is not given, the default is the currently active file (*AFD*): if file2 is not given, the default is *SINK* (i.e., the user's output stream). Since the normal use of $LIST is to produce a listing of a file for the user to read, the usual form of $LIST used is:

$LIST filename

which produces a listing of the file filename with line numbers in the output stream.

If filename is a line file, then the actual line numbers are printed in this listing, and the values of b, e, and i specified in the file name (or their default values) are used as described earlier. Thus,

```
$LIST X
```

lists line file X, starting at line number 1, using a line number increment of 1, and continuing to the end of the file. Lines in X with line numbers less than 1 will not be listed, nor will lines whose line numbers have fractional parts. Further, if Z is a line file, then:

```
$LIST Z(LAST)
```

lists the last line of Z, whereas:

```
$LIST Z(-5,5)
```

lists lines -5 through 5 of Z, while:

```
$LIST Z(0,10)+(50,100)
```

lists lines 0 through 10 and line 50 through 100 of file Z.

If a sequential file is listed, line numbers are printed in the listing as for line files: since sequential files do not have line numbers, these numbers clearly represent the order of the lines in the sequential file and not actual line numbers. However, one may specify b and e values for a sequential file in the $LIST command; if this is done, b must have value 1 or an error comment is given. The e value may be used as for line files, however. Thus, if S is a sequential file, then the command:

```
$LIST S
```

lists all of S, whereas the command:

```
$LIST S(1,10)
```

lists the first 10 lines of S only. A listing of a sequential file must always start with the first line of the file; the number of lines listed may be specified by the e value in the file name, however. Either file1 or file2 in a $LIST command may be a temporary file name; further, file1 may be explicit concatenation of file names. In addition, any implicit concatenation in the file listed is also effective; the listing will continue through all explicit and implicit concatenations of files producing one large listing of the several files. For example, if the file A ends with the line:

```
$CONTINUE WITH B(10,20)+C(100)
```

then the command:

$LIST A+D(LAST-10)

lists, in order, lines 1 through LAST of file A, lines 10 through 20 of file B, lines 100 through LAST of file C, and lines LAST-10 through LAST of file D. Files B, C, and D in this example must be line files, due to the use of starting line numbers other than 1; file A could be either a line file or a sequential file.

The $COPY command has the form:

$COPY [file1] [TO] [file2]

and produces a copy of file1, without line numbers, in file2. If file1 is not given, the default is the currently active file (*AFD*); if file2 is not given, the default is *SINK* (i.e., the user's output stream). The TO separator symbol may be optionally deleted from the $COPY command if the user desires. In the $COPY command, either file1 or file2 may be a temporary file. Also, either file1 or file2 may be an explicit concatenation of file names, and any implicit file concatenations in the files being copied are effective during the copy operation. The $COPY command may be used like a $LIST command to produce a file listing without line numbers; the various examples of $LIST commands in the previous paragraph are true for $COPY, also. That is,

$COPY Z(0,10)+(50,100)

is exactly the same as:

$LIST Z(0,10)+(50,100)

except that line numbers are not printed in the $COPY output listing. Copying files (rather than listing them) to the output stream is used in those cases in which the actual line numbers are not relevant; examples of this are listings of sequential files (which do not have line numbers) and listings of news files (see a later section) where only the line content is desired.

The more usual application of $COPY is to produce new files composed of the contents of one or more old files. Thus, the command:

$COPY A+B C

places in file C a copy of File A followed by a copy of file B, while the command:

$COPY A(1,10)+(20,40) D

places in file D lines 1 through 10 of file A followed by lines 20 through 40 of file A.

The command:

```
$COPY A(10)+B(5,15)+C D
```

places in file D lines 10 through LAST of file A, followed by lines 5 through 15 of file B, followed by lines 1 through LAST of file C, and the command:

```
$COPY A(50)+(1,49) B
```

places lines 50 through LAST of file A into file B, followed by lines 1 through 49 of file A. The $COPY command may be used to empty sections of a file. For example, the sequence:

```
-
-
-
$COPY A(1,10)+(20,50)+(100) -T
$EMPTY A
$COPY -T A
-
-
-
```

deletes lines 11 through 19 and lines 51 through 99 of file A. Note that sequential files cannot be handled in this fashion since one cannot specify any beginning line number other than 1 for a sequential file.

In a copy operation, the b, e, and i values given for the file file1 in the $COPY command determine which lines are to be copied; the b, e and i values given for file file2 determine the line numbers given to these copied lines as they are placed in file2. That is, in the command:

```
$COPY A(50,100) B(10,,10)
```

lines 50 through 100 of file A are copied into file B, where they are numbered in increments of 10 starting with line number 10. Hence, the first line from A is numbered 10 in B, the second line is numbered 20 in B, and so on. If no b and i values are given for file2, then the defaults of 1 and 1 are used. If file2 is a sequential file, then no line numbers are used and b, e, and i values are meaningless. One use of this feature is to clean up line files, after they have been modified. As a line file is edited, with lines deleted, replaced, and added, the file tends to become quite fragmented on a disc. Eventually, due to much editing, a line file may occupy much more disc space than it actually needs for its current contents, and the line numbers will not be sequential due to the editing. To clean up the file and renumber the lines, a sequence of the form:

```
-
-
-
$COPY file -T
$EMPTY file
$COPY -T file(b,,i)
-
-
-
```

may then be used. The e value, when given for file2, indicates when to stop placing lines in file file2 and may be used to copy into several files at the same time. For example, the command:

```
$COPY A B(1,10)+C(2,40,2)+D(100,,5)
```

places the first 10 lines of file A into file B (with line numbers 1, 2, ..., 10, in file B), the next 20 lines of file A into file C (with line numbers 2, 4, 6, ..., 40, in file C), and the remainder of the lines of file A are placed in file D (with line numbers 100, 105, 110 ...). Note that both file1 and file2, in the $COPY command, may be explicit file concatenations. The various b, e, and i values in file1 then determine which lines are copied; the various b, e, and i values in file1 then determine which lines are copied; the various b, e, and i values in file2 determine which files the copied lines go into and what line numbers are given to the copied lines.

When line files and sequential files are involved in a copy operation, some conversion is performed. If a line file is copied into a sequential file, then all line numbers are lost. If a sequential file is copied into a line file, then the line numbers specified by the b and i values in the line file name are used to number the lines as they are placed in the line file. In all cases, lines are trimmed as they are copied from one file to another. Note that when file 2 is a line file, only those lines of file2 copied into are changed; the rest of file2 is unchanged. If file2 is a sequential file, then lines copied into it are added at the end of the file; the contents of file2 before the copy operation are not changed.

As mentioned earlier, when a file is listed or copied, an implicit concatenation is effective if encountered. Thus, if file A terminates with the line:

```
$CONTINUE WITH B
```

then a $LIST or a #COPY command applied to file A will include file B, also. If, however, one wished to list or copy only file A, and not file B, one could first issue the command:

```
$SET IC=OFF
```

The value of IC controls implicit concatenation in MTS. When IC has the value ON, implicit concatenation is effective as described earlier; when IC has the value OFF, implicit concatenation is inhibited and a $CONTINUE WITH line is treated like any other file

line. Hence, the sequence:

```
-
-
-
$SET IC=OFF
$COPY A C
$SET IC=ONN
-
-
-
```

places a copy of file A into file C but inhibits implicit concatenation. If the last line of file A is:

```
$CONTINUE WITH B
```

then, after the $COPY command, the last line of file C is the same. The setting of IC has no effect upon explicit concatenation which is always effective.

In addition to the line number values (b, e, and i), file names may have certain modifiers attached to them also. These modifiers are discussed earlier; here we will briefly mention only two of them. A modifier is attached to a file name by preceding the modifier name with a @ and placing it at the end of the file name (after the b, e, and i values, if they are given). The two modifiers which we will look at here are the TRIM and the I (indexed) modifiers. As stated earlier, lines are trimmed as they move through the MTS I/O interface; that is, all but one trailing blank is deleted from each line. This is because the TRIM mode is always on within MTS. To stop this trimming, one may add @¬TRIM to a file name; the ¬ symbol negates the modifier and indicates that trimming should not be performed. Thus, to enter a source program into a file without trimming the lines, one could use the sequence:

```
-
-
-
$CREATE PROGRAM
$RELEASE
$EMPTY PROGRAM
$COPY *SOURCE*@¬TRIM PROGRAM@¬TRIM
 source program
$ENDFILE
-
-
-
```

Here, the $COPY command copies from the source stream until it finds an EOF (caused by the $ENDFILE). The source stream is read untrimmed and the lines are copied into PROGRAM untrimmed. Note that both files must have the @¬TRIM modifier attached. The @I modifier

causes a file to be read or written in indexed mode rather than in sequential mode; this modifier cannot be attached to a sequential file name. In the command:

```
$COPY A B@I
```

the lines to be copied are read in sequential mode from file A and written in indexed mode into file B. The line numbers used in this indexed write to file B are the line numbers of the copied lines in file A; thus, the lines placed in file B will have the same line numbers which they had in file A. The result is an exact copy of file A placed in file B, including line numbers. This is quite useful in those cases where a temporary change in a file is desired. One may copy the file into a temporary file maintaining the line numbers, edit the temporary file using a prior file listing, and then use the temporary file as edited. The temporary file may then be destroyed, or, if the user wishes to make the changes permanent, copied back into the original file. It is good practice, where possible, to be sure that changes are properly made in a file before actually destroying or changing the permanent copy of that file. As another example, if file L is a line file and file SL is a sequential-file-with-line-numbers, then:

```
$COPY L SL@I
```

copies the lines of file L, with their line numbers, into file SL, and:

```
$COPY SL L@I
```

restores line file L with the original line numbers.

Placing Information Into Files

To place source lines in a file, the user may make the file the currently active file and place the desired data lines in the source stream. There are two commands which can be used to make a file the currently active file. The $CREATE command discussed earlier creates a file and makes the created file the currently active file. Any subsequent data lines in the source stream will then be placed in this file, until the currently active file is changed. Also, any references to the pseudo-device *AFD* will be treated as references to the currently active file; one may therefore reference the currently active file without explicitly giving its name. The second MTS command which affects the currently active file is the $GET command:

```
$GET FDname
```

This command makes the file or device named FDname the currently active file or device; FDname must already exist in the system or must be a temporary file which will be created if it does not already exist. Whenever a $CREATE or a $GET is executed by MTS, the pertinent file becomes the currently active file, *AFD* is set to the pertinent file, and the file is opened (i.e. is prepared for I/O). Thereafter, data lines are placed in the file as MTS encounters them. To ensure that unwanted lines are not placed in a file, it is advisable to terminate a file as the currently active file after the desired data lines have been inserted into the file. This may be done with the command:

```
$RELEASE
```

which releases the currently active file. Subsequent data lines will be flagged by MTS as erroneous since it has no place to put them; references to *AFD* will be treated as references to an undefined file after a $RELEASE since there is no currently active file. The next $GET or $CREATE will, of course, re-establish a currently active file, which may be the same file as before or a different file.

Placing Information Into a Currently Active Line File

If the currently active file is a line file, then each data line found by MTS in the source stream must have a line number. MTS then replaces the line in the currently active file which has that line number by the line in the source stream. If a line is found which has no line number, there are two ways to provide a line number for a data line: the user may provide the line number with each line by typing the number at the beginning of the line; or the user may request MTS to automatically number each data line as MTS encounters the lines.

To manually provide a line number on a data line, the line number is typed starting in the first column of the line and is terminated by a comma; the line content then starts in the first column after the comma. Hence, the data line:

```
73,25    A=SQRT(B+C)
```

in a source stream causes line number 73 of *AFD* to be changed to:

```
25    A=SQRT(B+C)
```

Note, that this replaces line number 73 of *AFD*; if there was no line number 73 in *AFD*, then this data line is properly inserted in the file. If there was already a line number 73 in *AFD*, then this data line replaces the previous file line. Hence, this technique is used to modify a file as well as to place information in the file initially. (Note, also that if *AFD* is a sequential file and SEQFCHK is OFF, then the source line, independent of a line number, is placed at the end of *AFD*. One cannot replace lines in a sequential file, nor are line numbers meaningful in sequential files.) The comma which terminates the line number is called the Line Number Separator (LNS) character; it is taken as a separator only and is not part of the line number or of the data line content. One may change the LNS character by issuing the command:

```
$SET LNS=character
```

where "character" is the single character which the user wishes to use for the LNS. Thus, after the command:

```
$SET LNS=?
```

the previous data line would be typed as:

```
73?25    A=SQRT(B+C)
```

Unless the user issues a $SET command to change the LNS character, MTS assumes the LNS character to be a comma. The line number on a data line may be terminated by characters other than the LNS; this is fully discussed in the MTS Manual. For beginning users, the LNS should always be used to prevent a possible misinterpretation of a data line by MTS. A common error in typing replacement data lines for files is to forget to type the line number at the front of the line. In the above case, if one typed:

```
25    A=SQRT(B+C)
```

omitting the line number, then the 25 would be taken as the line number and the remainder of the line as the data line content, with the obvious erroneous results. On the other hand, if the data line:

```
100,    CALL SUBR
```

were erroneously typed as:

```
CALL SUBR
```

MTS would treat the line as an invalid command. In a conversational mode run, a file line may be deleted from *AFD* by typing a line number, terminated by a comma, and followed by a zero-length line. Thus, the data line:

```
175,eol
```

where eol is the end-of-line control sequence for the user's terminal, deletes line number 175 from *AFD*. Unfortunately, there is currently no way in MTS to provide a zero length source line in the batch mode, since at least one blank is always left on the line. Therefore, one cannot delete a line from a file in batch mode by this technique; there are other ways to delete file lines, however, which do work in batch mode (see later). As a final example, the following sequence may be used to replace lines 1, 13 and 23 of line file SOURCE:

```
-
-
-
$GET SOURCE
1,new line 1 content
23,new line 23 content
13,new line 13 content
$RELEASE
-
-
-
```

As for sequential files, the data lines in the above examples are trimmed before they are placed in *AFD*; that is, all but one trailing blank is deleted from each data line.

Automatic Numbering of Data Lines

When the user wishes to modify a few lines in a line file, he may proceed as in the previous paragraph, placing a line number on each data line. Clearly, however, if one wishes to enter a large amount of data into a file, it is most inconvenient to place a line number on each line when it is typed or keypunched. MTS therefore allows the user to request automatic numbering of data lines; this request is made by issuing the command:

```
$NUMBER [b][,i]
```

where b and i, if given, are decimal integers. If given, b is the first line number generated; if b is not given, it is assumed to be 1. If given, i is the line number increment used in generating line numbers; if not given, i is assumed to be 1. MTS then numbers the next data line as line number b, and the following data lines are numbered b+i, b+2i, b+3i, ..., and so on.

The data lines are then typed with no line number and with no LNS; MTS itself generates the necessary line numbers. In conversational mode, MTS types the next line number on the terminal, and the user responds by typing the data line and returning it to MTS. In batch mode, MTS simply swallows the data lines, numbering each line and placing it in *AFD*. Each data line, in either mode, replaces the line in *AFD* with the same line number. The forms of the $NUMBER command are then:

1. $NUMBER
   generate line numbers starting with line number 1 and using a line number increment of 1 (1, 2, 3, ...):
2. $NUMBER 10
   generate line numbers starting with line number 10 and using a line number increment of 1 (10, 11, 12, ...):
3. $NUMBER ,3
   generate line numbers starting with line number 1 and using a line number increment of 3 (1, 4, 7, ...):
4. $NUMBER 10,5
   generate line numbers starting with line number 10 and using a line number increment of 5 (10, 15, 20, ...):

The automatic numbering of lines continues until either a $UNNUMBER command or a $RELEASE command is issued; automatic line numbering then terminates. If the automatic line numbering is terminated by a $RELEASE command, then it will immediately restart after the next $GET or $CREATE command (i.e., as soon as *AFD* is again defined); in this case, re-numbering starts with the first previous unused line number and with the same line number increment. If the automatic line numbering is terminated by a $UNNUMBER command, then it does not restart until the next $NUMBER command is issued. In this case, one may type:

$NUMBER CONTINUE

to restart automatic line numbering with the first previous unused line number and with the previous line number increment, or one may specify new values for b and i as described above. During automatic line numbering, any MTS commands encountered in the source stream are executed; automatic line numbering continues after the command execution except for the $RELEASE and the $UNNUMBER commands. In particular, if another $NUMBER command is issued during automatic line numbering, the new values of b and i are immediately effective, and automatic line numbering then continues with these new values. To place a source program in a line file named PROGRAM, one could use the sequence:

```
-
-
-
$CREATE PROGRAM
$EMPTY PROGRAM
$NUMBER 10,10
 source program
$RELEASE
-
-
-
```

The first line of the source program in file PROGRAM will have line number 10; subsequent lines will be numbered 20, 30, 40 and so on. If the file PROGRAM already exists, one could use the command:

```
$GET PROGRAM
```

in place of the command:

```
$CREATE PROGRAM
```

in the above sequence. In the above cases, the new source program replaces the old contents of file PROGRAM due to the $EMPTY command, which empties the file before the new source program is placed in it. If one wished to add additional source statements at the end of line file PROGRAM, one could use:

```
-
-
-
$GET PROGRAM
$NUMBER LAST+10,10
 additional source lines
$RELEASE
-
-
-
```

The symbol LAST, when used in a $NUMBER command, is interpreted by MTS as the last line number of the currently active file. The above sequence thus places the additional source lines at the end of file PROGRAM, using line numbers n+10, n+20, n+30, ..., where n is the last line number of PROGRAM before the additions are made. If on the other hand, one wished to place two source programs in line file PROGRAM, one starting at line number 100 and the other starting at line number 200, one could use the following sequence:

```
-
-
-
$CREATE PROGRAM
$EMPTY PROGRAM
$NUMBER 100
 first source program
$NUMBER 200
 second source program
$RELEASE
-
-
-
```

In this sequence, the first source program starts at line number 100 of PROGRAM, and the second source program starts at line number 200. It is the user's responsibility, in such sequences, to ensure that the source programs do not overlap; in the above case, if the first source program is longer than 100 lines, then it will extend beyond line number 200. In this case, the second source program, starting in line number 200, would replace that part of the first source program which extended beyond line number 100. As before, all lines in the above examples are trimmed before they are placed in the currently active file.

Putting Data Into a Currently Active Sequential File

If the currently active file is a sequential file, then MTS will balk when a data line appears. This occurs because the operation of placing a data line into a file is an indexed write operation, and indexed I/O cannot be performed on sequential files. Therefore, if a sequential file is made the currently active file by a user, MTS must be informed that it is to place data lines into this file by a sequential write operation. To do this, one issues the command:

```
$SET SEQFCHK=OFF
```

which turns off the internal file checking in MTS and causes data lines to be sequentially written onto a sequential currently active file. Each data line is added to the end of *AFD* so long as this mode continues. To revert to the file checking mode, one issues the command:

```
$SET SEQFCHK=ON
```

after which data lines will cause error comments if *AFD* is a sequential file. Thus, to create a sequential file and place a source program in it, one could use the sequence:

```
-
-
-
-
$CREATE SOURCE TYPE=SEQ
$EMPTY SOURCE
$SET SEQFCHK=OFF
 source program
$RELEASE
$SET SEQFCHK=ON
-
-
-
```

If the file already existed, however, the $CREATE command in the above sequence could be replaced by the command:

```
$GET SOURCE
```

with the same effect, otherwise. If one wished to add source lines to the end of an existing sequential file, without changing the current contents of the file, one could remove the $EMPTY command from the above sequence:

```
-
-
-
$GET SOURCE
$SET SEQFCHK=OFF
 additional source program lines
$RELEASE
$SET SEQFCHK=ON
-
-
-
```

In any of the above cases, the source lines will be trimmed of all but one trailing blank before they are placed in the currently active file.

Placing MTS Commands in a File

There are times when one wishes to place MTS commands into a file. For example, it is possible to place a sequence of commands in a file and have MTS execute this sequence; this is described in the MTS Manual. Another example is the placing of $ENDFILE or $CONTINUE WITH lines into files as discussed in an earlier section. To do this, MTS provides the convention mentioned earlier in this section: if a source line begins with two dollar signs ($$), then it is treated as a data line and one $ is deleted. Thus, the data line:

```
293,$$ENDFILE
```

causes the data line:

```
$ENDFILE
```

to be placed in *AFD* as line number 293, whereas the source line:

```
293,$ENDFILE
```

causes an EOF in the source stream and does not affect *AFD*. To illustrate further, let us suppose that a user has a set of FORTRAN IV source programs in files named A, B, and C, and that he wishes to compile these as a single source program. He may use the command:

```
$RUN *FORTG SCARDS=A+B+C
```

to do this, using explicit file concatenation in the $RUN command. If, however, the files B and C are always used together, he may instead use implicit concatenation; the sequence:

```
-
-
-
$CREATE Z
$EMPTY Z
$NUMBER
$$CONTINUE WITH A RETURN
$$CONTINUE WITH B RETURN
$$CONTINUE WITH C RETURN
$RELEASE
-
-
-
```

creates a file Z which may be used to reference the three files A, B, and C, in that order; the $RUN command may be issued as:

```
$RUN *FORTG SCARDS=Z
```

Another sequence which does exactly the same thing as the above sequence is the following:

```
-
-
-
$CREATE Z
$EMPTY Z
1,$$CONTINUE WITH A+B+C
$RELEASE
-
-
-
```

Obviously, the user may employ any desired combination of explicit and implicit file concatenation to form logical files from individual files.

6.0 FILE ROUTINES

Sequential Files

Associated with every sequential file are at least three logical pointers which determine where the next read or write operation will start. That is to say, every sequential file has one Read Pointer for every file,[1] device usage block (FDUB) the user has attached to the file, exactly one Write Pointer and exactly one Last Pointer. These logical pointers are automatically updated after every read or write operation by the file routines as outlined below. In addition, there exists in MTS two FORTRAN callable subroutines, NOTE & POINT, whereby the user can "remember" the current values of these logical pointers and at some later time "alter" the values of these pointers. In so doing a user will be able to start reading and/or writing a sequential file from points other than the beginning and/or the end of the file.

As concerns the manipulation of these three logical pointers by the file routines, the following is the case. The Read Pointer is always initially set to point to the beginning of the file when the file is created. This is also the case for each use of the file (i.e., every time the file is opened). The Read Pointer is updated after every read operation to point to the next line to be read. Finally the Read Pointer is reset to point to the beginning of the file when ever the file is emptied or rewound. The Write Pointer is initially set to point to the beginning of the file when the file is created, and is updated after every write operation to point to the next line to be written. The Write Pointer is reset to point to the beginning of the file when the file is emptied or rewound. In addition, if after any read operation, the Read Pointer is updated past the Write Pointer, the Write Pointer is updated to coincide with the Read Pointer. This allows a user to rewind a sequential file, begin reading, stop at some intermediate point and begin writing at that point. This is similar to what would happen if the same operations were performed on a tape. The difference being that if after writing a few lines, the user again began to read the file, he would begin reading from the intermediate point at which he previously stopped reading and started writing. (That is to say that the Read Pointer is not updated after a write operation).

[1] When a user has more than one logical I/O unit attached to the same file, MTS creates a FDUB for each I/O unit, and each FDUB points to the one file control block (FCB) for the file. (That is not the case when two separate users each have a logical I/O unit attached to the same (shared) file. In this case each user has a FDUB pointing to his own FCB for the given file.) The ramifications of more than 1 Read Pointer will be discussed later.

Finally, whenever the file is opened, the Write Pointer is set equal to the Last Pointer.

The Last Pointer is initially set to point to the beginning of the file when the file is created, and is updated after every write operation to coincide with the new updated Write Pointer. The Last Pointer is also considered the logical end of file, so that writing a file beginning from some intermediate point implies that any information from that point on is to be discarded. (It is planned in the not too distant future to provide a replace facility for sequential files; however, for now, the above will always be the case.) Finally, the Last Pointer is reset to point to the beginning of the file, whenever the file is emptied.

As concerns the MTS NOTE & POINT subroutines, the following can be said. As indicated by the attached subroutine descriptions, after calling NOTE, four fullwords of information are returned to be remembered (saved?), to be used later as the caller dictates. These four fullwords are respectively, the Read, Write and Last Pointers, as well as the last line number (useful only for sequential files with line numbers)[2] associated with the file corresponding to the FDUB given. These pointers always correspond to the next line about to be read or written.

At some later point in time, the caller is able to indicate which of these pointers he wished to alter for the next read or write operation and using the information returned by NOTE, can call the POINT subroutine appropriately.

Finally, the user should be sure he understands the consequences of reading and/or writing multiple logical I/O units attached to the same file. Since there is one Read Pointer associated with each FDUB and thus with each logical I/O unit, the user is able to read alternately from a number of different points in the file, overlapping or not as the application dictates. However, since there is only one Write Pointer and one Last Pointer associated with the file, writing to multiple logical I/O units attached to the same file, amounts to simply appending to the end of the file in the order that the write requests are received.

The following is of interest primarily to assembly language users. Since sequential files can have lines (logical records) of up to 32,767 bytes in length, before trying to read a file, GDINFO should be called to find out what type the file is, and also what the maximum logical record length is.

[2] The rational for wanting to remember the last line number in a sequential file with line numbers is that when writing this type of file, the caller must insure that line numbers are in numerically increasing order.

When calling the I/O subroutines SCARDS, SPRINT, READ, WRITE, etc., the user should be sure he understands the use of the indexed and sequential modifer bits given in the calling sequence. In general, these modifiers refer to whether line numbers are being provided by the caller or should be generated by MTS. Thus, for example, to write a sequential with line number file, the indexed modifier should be turned on to write the line with the desired line number, otherwise MTS will generate a line number on it's own (not necessarily greater than the line number of the previous line in the file). Furthermore, when reading a sequential file, with the sequential modifier on, MTS will always return a line number, either the one associated with the line if the file is of type sequential with line numbers, or a generated one. The indexed modifier should never be on when reading a sequential or sequential with line number file.

The following is of interest primarily to tape users. Tape users should realize that sequential files are very similar to tape files, consequently, most applications which use tapes, could with slight modification use sequential files. Obviously, such operations as forward spacing files, writing end of file marks, etc., do not have meaning concerning sequential files. (It is planned in the not to distant future to impliment forward and backward space record facilities as well as a read backward sequential capability for files.) Furthermore, the @cc modifier bit is completely ignored by the file routines; thus, for instance, files must not be rewound by writing the three character record REW into the file with the @cc modifier bit on. The subroutine REWIND should always be called to rewind a file (it will also rewind tapes, of course). Finally, it should be noted that blocking of records is not nearly as space saving on a sequential file as on a tape file since there are no "gaps" between logical records in a sequential file. However, blocking does improve the efficiency of sequential files quite a bit, since the overhead of going through the MTS interface is reduced.

7. INTERNAL FILE STRUCTURES

This section will be added, in the form of an update, at a later date.

*CATALOG

Contents: The object module of a program to print a list of file names for the current user.

Usage: This file should be referenced by a $RUN command with *CATALOG as the object file.

Logical I/O units referenced:
SPRINT - the list of files

Examples:
```
$RUN *CATALOG
$RUN *CATALOG PAR=3/LINE,SEQ
$RUN *CATALOG PAR=3,S
```

Parameters: By the use of the PAR= field in the $RUN command, a selected subset of the user's files can be listed. The legal parameters are divided into three groups At most one from each group may be specified; if not specified, the default given will be used. Abbreviations for terms are in parenthesis.

1. Type
   a. LINE (L) - all line files
   b. SEQ (S) - all sequential files
   c. SEQWL - all sequential-with-line-numbers files
   Default - all types of files belonging to the user

2. Kind
   a. TEMP (T) or SCRATCH (SC) - all the user's temporary scratch files
   Default - all the user's private files

3. Format
   3/LINE (3) - output format is three items per line
   Default - one item per line

Thus the second example above would list all the user's files that were of sequential type and would print the output three items per line. The third example is the same as the second, except using the abbreviations.

*FILEDUMP

Purpose: To dump a magnetic tape or file in both character and binary formats.

Prototype: $RUN *FILEDUMP O=*pseudodevicename* [PAR=parameters]

Example: $RUN *FILEDUMP O=*TAPE* PAR=FILES=3,EBCDIC

Logical I/O units referenced:
SPRINT - dump output
O - magnetic tape or file to be dumped

Parameters:
FILES=n
RECORDS=m
EBCDIC
BCD
NODUMP

Description: *FILEDUMP will dump a specified number of files and records in either BCD and octal or EBCID and hexadecimal. The number of files and records to be dumped as well as the format are controlled by the parameter field of the $RUN command.

Specifying "FILES=n" and/or "RECORD=m" will cause n files and m records to be dumped. If only a"FILES=n" parameter is given, then n complete files will be dumped. If both "FILES=n" and "RECORDS=m" are given, then n files plus the first m records in the file n+1 will be dumped. If neither of these parameters is given, then the tape will be dumped until two consequentive filemarks are encountered.

"BCD" or "EBCDIC" will force the dump to be in the appropriate format. If BCD format is requested, the two high-order bits of each data byte are masked off in the dump. All non-printing graphics in the EBCDIC (or BCD) portion of the dump will print as the character period(.).

The parameter NODUMP will suppress the printing of the dump portion of *FILEDUMP output. The header line which precedes each record and which gives the file and record numbers, the record length, density and mode will continue to print, but the data dump will not.

If the FDname specified as logical device zero is a 7-track tape, the default is BCD. If the FDname refers to a file or a 9-track tape, the default is EBCDIC.

The BCD character set uses the IBM scientific (ALTERNATE) BCD graphics. Note that this character set is not the IBM STANDARD (BUSINESS) BCD.

*FILESCAN

Contents: The object module of the file scan program.

Purpose: To locate a line in a file according to a given format and print the line and line number.

Usage: The filescan program is invoked by an appropriate $RUN command specifying *FILESCAN as the file where the object cards are found.

Logical I/O units referenced:
SCARDS - format and file name input.
SPRINT - requests, line, and line number of selected line.
SERCOM - error comments.

Description: *FILESCAN will first print "ENTER FORMAT" on SPRINT. There are three types of format terms: skip, transfer, and character string. These are entered via SCARDS. The skip term consists of the letter S followed by an optional minus sign followed by a string of decimal digits. The value of the digit string is the number of columns to be skipped. The transfer term consists of the letter T followed by a string of decimal digits. The value of the digit string is the column in the lines of the file to be scanned where the next skip or character string term will start. The character string term is delimited by primes. The characters in the string are compared with the appropriate columns of the lines of the file to be scanned. A maximum of 20 character strings may be used in one format. If a prime is desired in the character string, it may be represented by two consecutive primes. (Note: Unless it is the last character in a character string, each prime reduces the maximum number of character strings by one.) Commas may be used to separate format terms, but they are not necessary. The format is terminated by the first blank not in a character string or by the end of the line. If the first character entered is a >, it is assumed that the characters immediately following name a file and that the previous format is to be used. If "CONTINUE" (or "C") is entered, the scan is continued with the next line in the same file using the same format.

After the format is entered, *FILESCAN will print "ENTER FILE NAME" on SPRINT. The file name is entered via SCARDS and may be followed by a line number range and/or modifiers as for GETFD. The first blank or end of line terminates the file name. Lines in the file must be less than 256 bytes in length.

An end-of-file when a file name is requested will cause a format request. An end of file when a format is requested will cause termination.

Example: If it was desired to find lines containing "LA" in columns 10 and 11 and "R4," in columns 16-18 of the files FILE and FILE1, the input (underlined) and requests might be:

```
#$RUN *FILESCAN
#EXECUTION BEGINS
  ENTER FORMAT
 T10'LA',S4'R4,'
  ENTER FILE NAME
 FILE
     114                LA     R4,INPUT
  ENTER FORMAT
 CONTINUE
     123                LA     R4,1(,R4)
  ENTER FORMAT
 >FILE1
      12                LA     R4,=A(WHERE)
  ENTER FORMAT
 ENDFILE
#EXECUTION TERMINATED
```

*FILESNIFF

Contents: The object module of the file-sniffing program.

Purpose: To print the storage characteristics of MTS files.

Usage: The program accepts the names of files and displays the characteristics of each. (No change is made to any file). If information on only one file is wanted, its name should be given as the parameter field (PAR=) on the $RUN command. If the PAR field is omitted, names are read via GUSER until an end-of-file is sensed.

Logical I/O Units Referenced:
SPRINT - the output describing the file.
GUSER - the names of the files, one at a time.
SERCOM - the message "ENTER FILE NAME" before reading each file name.

Examples:

```
#$RUN *FILESNIFF PAR=TEST3
#EXECUTION BEGINS
     FILE=TEST3
          LOC=DISK
          TYPE=LINE
          VOLUME=MTS005, NO.  EXTENTS=02
          NO.  LINES=00465, AVE.  LENGTH=080
          NO.  PAGES=(014,012), SIZE PARS=(01024,00896).
          PERMIT CODE=NONE
#EXECUTION TERMINATED

#$RUN *FILESNIFF
#EXECUTION BEGINS
     ENTER FILE NAME...
ROOT
     FILE=ROOT
          LOC=DISK
          TYPE=SEQUENTIAL
          VOLUME=MTS101, NO.  EXTENTS=01
          NO.  PAGES=(002,002), SIZE PARS=(00128,00128).
          PERMIT CODE=RO
     ENTER FILE NAME...

[enter an end-of-file to terminate]

To run *FILESNIFF on all your files:
$RUN *CATALOG SPRINT=-TEMP
$RUN *FILESNIFF GUSER=-TEMP
```

Description of output:

LOC=	The device on which the file is located.
TYPE=	The type of file organization.
VOLUME=	The name of the direct access volume on which the named file is located. If it is divided among several volumes, this is the name of the volume on which the first extent resides.
NO. EXTENTS=	The number of distinct contiguous areas on the named volume (and possibly other volumes) allocated to the file.
NO. LINES=	The current number of lines in the file (only for line files).
AVE. LENGTH=	The average number of characters per line (only for line files).
NO. PAGES=(X,Y)	X is the current number of pages of storage (for the specified file) for which the user is charged. Y is the number for which he would be charged if the file were re-created with a size appropriate for its current contents.
SIZE PARS=(W,Z)	W is the "current size parameter" for the file. Z is the "minimal size parameter". W roughly describes the size of the file in terms of the optional SIZE parameter on the CREATE command, and Z is the size parameter which should be given if the owner wishes to recreate the file with minimal size appropriate for its current contents.

*FILEUSE

Purpose: To print out the use count and last reference date as well as other information about a file.

Usage: The program is invoked by a $RUN command.

Logical I/O units referenced:
GUSER - Input for file names if no parameter given.
SERCOM - Commands to user
SPRINT - Output information

Parameters: Filename or *

See description below

Examples:
```
$RUN *FILEUSE PAR=AFILE
$RUN *FILEUSE PAR=*
```

Description: If a parameter is given, the program checks for * in which case all the user's files are assumed respectively; otherwise the user file name given is used.

If no parameter is given, the user is asked via GUSER to enter a file name. An end-of-file will terminate the program at this point.

The single line of output printed on SPRINT consists of the following:

- FILENAME -
- OWNER - This is the signon-ID of the person who created the file.
- TYPE - LINE, SEQ or SEQW
- LOCATION - DISK
- USECOUNT - An integer count of the number of times the file has been used ("opened") since the file was created
- LASTREF - The date the file was last referenced ("opened")
- CREDATE - The date the file was created
- VOLUME - The name of the direct access device on which the file resides.

Note: Such things as $RUNning, $LISTing and $COPYing a file as well as *EDITing, *FILESNIFFing, etc., will change the use count last reference date. However, the saving of files performed by the system nightly does not affect these counts, nor does the running of *CATALOG or *FILEUSE.

*FSAVE

Contents: *FSAVE contains the object module of the file save and restore program.

Purpose: To save files (sequential or line) on tape and restore them from tape by name with line numbers.

Usage: The program is invoked by a $RUN command.

Logical I/O units referenced:

SCARDS - The source of instructions to *FSAVE
SPRINT - Program messages and error comments.
SERCOM - error messages
GUSER - prompting, if conversational
0 - The pseudo-device name of the 9 track tape to be used.

Description: *FSAVE reads its instructions from SCARDS, prints messages and error comments on SPRINT, and prompts on GUSER and SERCOM.

The lines from SCARDS contain commands (followed by the filenames which the commands are to operate on in the case of save and restore). The commands are preceded by three dots (...) to distinguish them from file names. One letter abbreviations of the commands are allowed.

The commands are:

1. ...SAVE (abbreviation: ...S)

   This command causes *FSAVE to save the files whose names are given on lines following the "..SAVE" command. If the line contains one name, the file by that name is saved on the tape and remembered on tape by the same name. If the line contains two names, the first name is taken as the file name on disk but the file is labeled on tape with the second name.

   Do not specify a line number range on any of the file names. *FSAVE will save the entire contents of the file (from line -99999.999 to +99999.999). Each line will retain its original number, if restored to a line file.

2. ...RESTORE (abbreviation: ...R)

   This command causes *FSAVE to restore the files whose names are given on lines following the "...RESTORE"

command. If the line contains one name, the file on tape is restored to a file on disk by the same name.

If the line contains two names, the first name is the name of the file on tape and the second is the name of the disk file into which it is to be restored. *FSAVE empties the disk file before restoring. If ...RESTORE is followed by ...ALL, all files on the tape will be restored. If the disk file is not the same type as the file originally saved, this will be noted on SPRINT and the user will be given the opportunity to cancel the request in terminal mode.

3. ...LIST (abbreviation: ...L)

This command causes a list of all the files currently on this tape to be produced.

Notes:

*FSAVE maintains a table of contents at the end of the tape (it has to be at the end so that the tape can have data added to it --- a record at the beginning of the tape cannot be safely overwritten). The first thing *FSAVE does is skip to the end of the tape and read the table of contents. At this point, the tape is positioned correctly for saving additional files. If you are going to save and restore files in the same run, it is more efficient to save first, then restore, since restore requires that the tape be rewound. The list function works from the in-core copy of the table of contents and can be executed at any time without penalty.

The first time a tape is used, you must specify PAR=INIT so that *FSAVE does not attempt to find a non-existent table of contents. When the same tape is later used, you must <u>not</u> specify PAR=INIT since this would cause *FSAVE to overwrite files that have been previously saved.

The maximum number of files that may be saved on a tape is 250.

The maximum number of letters in a file name is 16.

Parameters: PAR=INIT is specified if this is the first time that any files are to be saved on this tape.

Examples:
```
$RUN *MOUNT PAR=L999 9TP *TAPE* RING=IN 'TAPE ID'
$RUN *FSAVE 0=*TAPE* PAR=INIT
...SAVE
X
Y FILE2
```

```
Z
$ENDFILE
```

This run initializes a tape and saves the files X, Y and Z. The file Y is saved under the name FILE2.

```
$RUN *MOUNT PAR=P999 9TP *SAVE* RING=IN 'TAPE ID'
$RUN *FSAVE O=*SAVE*
...SAVE
Q
FORT SOURCE
...RESTORE
RFILE
OBJ2 XRQ
...LIST
$ENDFILE
```

This run uses a tape which has previously been used for file save. It adds the files Q and FORT (under the name SOURCE) to the tape, restores the files RFILE and OBJ2 (OBJ2 is restored to the disk file called XRQ), and lists the names of all the files on the tape.

*PERMIT

Contents: The object module of a program to allow users to permit their files to be accessed by other users.

Purpose: To allow files to be shared between user ID numbers.

Usage: The program is invoked by a $RUN command.

Logical I/O Units Referenced:
GUSER - filename, access type and/or sharers (unless parameters are given).

Description: Three parameters separated by blanks are required as input for each file being permitted. They are:

What How Who

What = the name of the file to be shared.

How = the type of sharing access allowed. Currently Read-Only is the only type of access allowed and is designated by RO. This parameter should be omitted if Read-Only is not desired.

Notes:
1. Designation of a file by a user as Read-Only shared allows other users to access that file directly (without having to copy it via the *COPY program). This file may then only be read - it may not be changed, even by the owner of the file.
2. If the owner of the file "X" has SIGNON ID ABCD, then the filename ABCD:X will refer to this file and may be used anywhere a filename may be used provided that the usage is a read-type usage and provided that user ABCD has permitted file "X" as Read-Only shared.
3. If one has a file whose name has a colon as one of the first five characters, he must refer to it using XXXX:filename where XXXX is his SIGNON ID ,e.g., if user ABCD has file 0:1 he must refer to it as ABCD:0:1.
4. The file should be completely prepared and checked before being permitted as RO. If changes must be made later, it must be unpermitted (permitted with NONE) before the changes can be made. This must not be done when someone else is using it. Pre-

sently there is no check made to determine whether someone is accessing the file when it is being unpermitted.

Who = the sharers who are allowed to access the file. Currently three classes of sharers are allowed. These are:

- ALL - meaning all users (default).
- PRJNO - meaning all users who have the same project number as the permitting user. The project number appears as "charge number" on the tail-sheet of batch runs. It is not (in most cases) the same as the SIGNON ID. Students in a class, or other groupings of users, are assigned different SIGNON ID's but the same project number.
- NONE - meaning no one else.

Notes:

1. Current restriction: If RO is specified, ALL is assumed.
2. If PRJNO or ALL is specified, the program *COPY may be run to make a copy of the file under a new user ID.
3. When a file is created, it has access type NONE.

Examples:

```
$RUN *PERMIT  PAR=MYFILE RO
```

permits the file to be Read-Only shared by all users.

```
$RUN *PERMIT  PAR=MYFILE2
```

permits the file to be copied by all users via *COPY.

To permit several files, the parameter field should be omitted. The program will request the information via logical I/O unit GUSER.

In either terminal or batch mode, the program will read lines, each having a file name, an access type and/or a sharing class, until an end of file is read.