SECTION VIII
Files

For problems that require permanent data storage external to a particular program, BASIC/3000 provides a data file capability. This capability allows flexible, direct manipulation of large volumes of data stored on files.

There are three types of files used in BASIC/3000: formatted files, binary files, and ASCII files. Formatted files are created and accessed through the BASIC/3000 Interpreter. Binary and ASCII files are created in MPE/3000 but can be accessed with BASIC/3000.

A catalog of ASCII and binary files, as well as of formatted files in the user's group library, can be requested with the BASIC/3000 CATALOG command (see Commands, Section II).

BASIC FORMATTED FILES

A formatted BASIC/3000 file is created under control of the BASIC/3000 Interpreter. It contains format words to indicate the type of the data items in the file. These format words are placed in each record automatically by the Interpreter (see Appendix H for formatted file structure). Formatted files allow run-time checking of the type of each data item.

BASIC formatted files are created with the CREATE command or statement. They may be accessed by any file statements or commands including UPDATE and ADVANCE, but not File LINPUT which is reserved for ASCII files. The user's ability to access a BASIC file depends on the MPE file security restrictions.

ASCII FILES

ASCII files are created through the MPE/3000 Operating system and treated by BASIC/3000 as terminal-like devices. They can be actual terminals. When the PRINT # statements described in this section are used, output is formatted according to the rules for the PRINT statement (see Section II), although the length of the print line for an ASCII file may be changed by using the MARGIN statement described in this section. Printing to ASCII files according to a specified format requires the use of the PRINT # USING statement (see Section IX). Input from ASCII files is analyzed according to the rules of the INPUT statement, except that reading starts with the next item, not the next record.

All file statements except ADVANCE, UPDATE, and CREATE may be used with ASCII files. See ASCII File Access in this section for restrictions on accessing ASCII files.

BINARY FILES

Binary files are unformatted files created through the MPE/3000 Operating System. Data items are stored in binary files as binary words without type information. When data is read from a binary file, it is assumed to be the type of the variable into which it is being read. Items in binary files can cross record boundaries and new records are used only upon overflow.

All file statements except ADVANCE, UPDATE, File LINPUT, and CREATE can be used with binary files. Access to binary files is discussed under Binary File Access in this section.

FILE NAME

When any file is created, whether it is ASCII, binary, or BASIC formatted, it is assigned a file name by the user who creates the file. The file name may contain up to eight alphanumeric characters, the first of which must be a letter. The file name may be fully qualified as follows:

file name [/lockword] [.group name [.account name]]

If a *lockword* was specified for the file at its creation, this same *lockword* must follow the *file name* when the file is accessed.

The *group name* and *account name* specify a group and account other than those under which the user logged on. If the file is part of the user's group and account, then these qualifiers may be omitted.

The unqualified file name specifies a file in the user's log-on group and account that is not restricted by a lockword.

Refer to the *HP 3000 Multiprogramming Executive Operating System Reference Manual* (03000-90005A) for details of the file name specification.

Creating a Formatted File

A formatted file can be created by the CREATE command or through the CREATE statement. CREATE allocates a file of a specified size, assigns a name, and initializes each record with an end-of-file mark. Both the record size and the number of records can be specified. Neither of these sizes can be changed later.

Form

CREATE command:

CREATE file name, file length

CREATE file name, file length, record size

file name is a simple string without quotes. The *file length* is an integer constant. The optional *record size* is an integer constant.

CREATE statement:

CREATE numeric variable, file name, file length

CREATE numeric variable, file name, file length, record size

The *numeric variable* is used to return a result about the status of the file. *File name* is a string expression, *file length* is an integer expression, and the optional *record size* is also an integer expression.

Explanation

The file name may be fully qualified by a lockword, group name, and/or account name.

The file length specifies the number of records to be allocated to the file.

The record size specifies the number of data words per record. Record size may be between 4 and 319 words; the default size is 106. The most efficient record sizes are 106, 212, and 319. Record size is the number of words needed by the user to contain his data; the BASIC/3000 Interpreter adds format words to each record (see Appendix H).

The numeric variable contains one of the following results when the CREATE statement is executed:

0	successful file creation
1	a file already exists with the same name
2	the file was not created for some reason other than a duplicate name

An error message will be printed if the CREATE command cannot create the specified file name because of a duplicate name or some other reason.

CREATE does not open the file for access. Files are opened with a FILES or an ASSIGN statement (see Opening Files, this section).

EXAMPLES:

```
>CREATE AFILE,30

  10 DIM B$[4]
  20 LET B$:"BB"
  30 CREATE N2,"BFILE",15,212
  40 CREATE N3,"XFILE",20
  50 CREATE N4,"AA",15
  60 CREATE N5,B$,15
  70 CREATE N6,"DFILE",20
  80 CREATE N7,"FF2",20
  90 CREATE N8,"XX",10
 100 PRINT "N2=";N2,"N3=";N3,"N4=";N4,"N5=";N5
 110 PRINT "N6=";N6,"N7=";N7,"N8=";N8
>RUN
N2= 0               N3= 0               N4= 1               N5= 0
N6= 0               N7= 0               N8= 0
```

Eight files are created, the file AFILE with a CREATE command, the remainder with CREATE statements. All but BFILE have the default record size 106; records in BFILE have 212 data words.

In this run, the value of N4 is 1 indicating that a file name AA already exists.

Note that the file name in the CREATE command is an unquoted string; in the CREATE statements, the file name may be a string expression such as a quoted string or a string variable.

Purging a File

An ASCII, binary, or BASIC formatted file can be deleted from the system with a PURGE command or PURGE statement.

Form

PURGE command:

PURGE file name

where *file name* is a simple string without quotes.

PURGE statement:

PURGE numeric variable, file name

The *numeric variable* will contain a result following execution of the PURGE statement. The *file name* is a string expression.

Explanation

The file specified in the statement or command is purged and is not recoverable.

The numeric variable in the statement returns a result on the status of the purge operation:

0	successful purge
1	file is being accessed and cannot be purged
2	user is not permitted to purge this file
3	there is no such file

EXAMPLES:

```
>PURGE AFILE
  10 PURGE N,"BFILE"
  20 PRINT N
>RUN
 0
```

A PURGE command is used to purge AFILE, a PURGE statement to purge BFILE. The result of purging BFILE is printed. Since it was a successful purge, the result is zero. If the PURGE command had been unsuccessful, a message would have been printed.

Opening Files

In order for a program to access a file, the file must be open. For every file that is to be opened, an association is established between the file number used in access statements and the file name. The file number is an integer between 1 and 16. The file name of a BASIC/3000 formatted file is the name assigned with a CREATE command or statement. The file name of an ASCII or binary file is the name assigned when the file was created under MPE/3000 control (see *MPE/3000 Operating System Manual* for instructions).

The linkage between file name and file number is accomplished by one of two statements: the FILES statement or the ASSIGN statement. FILES causes file numbers to be assigned to the files and, if a file name is specified, the file is opened. ASSIGN associates a file name with a file number reserved by FILES but not named. It opens a file not previously opened by FILES.

FILES is a declarative statement, not a dynamic statement. This means that it is not executed but is processed before the run begins. It may appear anywhere in the program.

ASSIGN, on the other hand, is a dynamic statement. It is executed during the program run and its position affects program execution. If ASSIGN is used to open a file, it must be executed before any statements used to access that file.

> *Note: Readers with experience using HP 2000 Time Shared Basic may be confused by the use of the term open as used for the BASIC/3000 Interpreter. OPEN for 2000 Time Shared Basic is equivalent to CREATE for BASIC/3000.*

CLOSING FILES

All files are closed automatically upon program termination. A file may be dynamically closed during program execution with the ASSIGN statement. This should be done wherever practical to release buffer space for other files.

FILES Statement

Every file that is to be accessed must have a file number designated in a FILES statement. Each file designator reserves a file number starting with number 1. Up to 16 file numbers may be reserved for any run. If the file designator names a file, the file will be opened.

Form

FILES file designator list

One or more *file designators* may be specified, separated by commas if more than one. A *file designator* may be one of the following:

file name

*

#integer

The *file name* identifies an existing file created with the CREATE command or statement, or an ASCII or binary file created in MPE/3000. It may be fully qualified.

The * reserves a file number for a file that will be named and opened with an ASSIGN statement.

#integer specifies the internal file number equivalent to an existing file number.

Explanation

File designators are associated with file numbers in the order in which they appear in the FILES statement. The first is assigned file number 1, the second file number 2, and so forth. If there is more than one FILES statement, file numbers are reserved starting with the first FILES statement.

When a file name is specified, the file is opened and the program is given both read and write access to the file unless the file is already open in some other program. In this case, only read access is allowed and a warning message will be printed at execution time. If the file cannot be opened, the program terminates.

When the program that opened a file terminates, the file is closed. If the program had read and write access to the file, the write restriction is removed. This enables the next program opening the file to have both read and write access.

When an asterisk (*) is used instead of a file name, the file number is reserved but the file to be associated with that number is not specified. The ASSIGN statement must be used to associate a file name with the file number. ASSIGN must be specified before any reference is made to the file number.

If *#integer* is used instead of a file name, it specifies an internal file number. This number identifies a file declared with a FILES statement in another program when programs are segmented. (See Section X, Segmentation.)

The FILES statement is declarative, not dynamic; it may appear anywhere in a program and is not executed.

EXAMPLES:

```
10 FILES AFILE,BFILE,*
20 FILES AA,BB,*,*,*
30 FILES #6
```

Five files are specified in the three FILES statement. The formatted files AFILE and BFILE have file numbers 1 and 2 respectively. Files AA and BB have file numbers 4 and 5 respectively.

The files reserved for numbers 3, 6, 7, and 8 must be assigned names in an ASSIGN statement before they can be accessed.

The file associated with local file number 9 in statement 30 has been previously associated with internal file number 6. For the relation between internal and local file numbers, see Files and Segmentation in Section X, "Segmentation".

ASSIGN Statement

The ASSIGN statement is used to assign a file name to a file number reserved by FILES and open the file. If another file was associated with the file number, that file is closed. The result of the open operation is returned following execution of ASSIGN. Unlike FILES, the ASSIGN statement is executed.

Form

The forms of ASSIGN are

ASSIGN file name, file number, numeric variable

ASSIGN file name, file number, numeric variable, mask

ASSIGN file name, file number, numeric variable, restriction

ASSIGN file name, file number, numeric variable, mask, restriction

*ASSIGN *, file number*

The *file name* is a string expression; the *file number* is an integer expression with a value between 1 and 16. The *numeric variable* returns the result of the ASSIGN execution. The optional *mask* is a string expression used to encode or decode file data. The optional *restriction* is a two-letter code to specify any access restrictions on the file.

Explanation

In the first four forms, the file name is associated with the file number and the file is opened.

If an * is used instead of the file name, any file previously associated with the file number is closed. If the file is already closed, the statement is ignored. Since closing a file releases the buffer space that was allocated to it, it is good practice to close unneeded files.

An error results if the specified file number exceeds the number of positions in the program's FILES statements.

After ASSIGN is executed, a value is returned to the numeric variable:

0	file is available for read and write
1	file is available for read only
2	(unused)
3	the file does not exist or is not accessible
4	(unused)
5	no buffer space is available for the file
6	file is not available for read or write because of another user's current access
7	specified restrictions not possible
8	file is available for write only (applies to "write-only" files, such as those directed to a line printer).

If the value returned is 3, 5, 6, or 7 the file is not opened and any access to the file number causes a fatal error. If the returned value is 1 any attempt to print onto the file causes a terminal error. If the returned value is 8, any attempt to read the file causes a terminal error. Other references to the file assigned that file number are legal.

A *mask*, if specified, protects the data in a file. Whenever the file is assigned the same mask should be used, or the data will not be intelligible when read. The actual data is scrambled or unscrambled using the mask; the data types and the end-of-file or end-of-record marks are not affected.

The *restriction* may be one of the following:

Code	Meaning
RR	Read and Write Restriction – no other user can access the file
WR	Write Restriction – other users may read, but not write on, the file
WL	Write Restriction with Dynamic Locking – current user has option to lock file; other users may read only
NR	No Restriction – current user does not have the option to lock file; other users can read from and print on the file
NL	No Restriction with Dynamic Locking – no restriction but user has option to lock the file
RD	Read Access Only – current user may read from, but not write on, the file; current user does not have option to lock file; other users can read from and print on the file
RL	Read Access Only with Dynamic Locking – current user may read from, but not write on, the file; no restriction but user has option to lock the file

If the *restriction* is omitted, the file is opened with WR restriction. If this fails, the NR restriction is used.

Ths specified restriction is placed on the file and remains in effect as long as the file is open. If another restriction is in effect due to a concurrent access, the ASSIGN statement will return the result 6, and the file is not opened.

Examples

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 30 ASSIGN "XFILE",3,X1,WR
 40 ASSIGN "DFILE",6,D1,RR
 50 ASSIGN "FF2",7,N,NL
 60 LET X$="X"
 70 ASSIGN X$,8,X,"ABZ1"
 80 PRINT X1,D1,N,X
 90 ASSIGN *,8
100 ASSIGN "CC",1,C1
110 PRINT C1
>RUN
 0                 0                 0                 0
 0
```

Files are assigned for each file number associated with an * in the FILES statements. A write restriction on XFILE prevents other users from writing on that file. A read and write restriction on DFILE prevents other users from having any access to that file. File FF2 can be locked and unlocked with the dynamic locking statements LOCK and UNLOCK; there are no access restrictions on FF2.

The Mask "ABZ1" is used to encode the data in file X. File X is closed in line 90.

In line 100, AFILE is closed and file CC is assigned file number 1 and opened.

The zeros in the numeric variables indicate that each file was available for reading and writing in the current run when it was opened.

The following example shows the values returned from an ASSIGN on the same file during the *same* program versus an ASSIGN on the same file from two *different* programs.

Same Program

1st ASSIGN \ 2nd ASSIGN	RR	WR	WL	NR	NL	RD	RL
RR	6	6	6	6	6	6	6
WR	7	7	7	0	3	1	3
WL	7	7	7	3	0	3	1
NR	7	7	7	0	3	1	3
NL	7	7	7	3	0	3	1
RD	7	0	3	0	3	1	3
RL	7	3	0	3	0	3	1

Different Programs

1st ASSIGN \ 2nd ASSIGN	RR	WR	WL	NR	NL	RD	RL
RR	6	6	6	6	6	6	6
WR	7	7	7	1	3	1	3
WL	7	7	7	3	1	3	1
NR	7	7	7	0	3	1	3
NL	7	7	7	3	0	3	1
RD	7	0	3	0	3	1	3
RL	7	3	0	3	0	3	1

File Access

There are two types of access to a file: serial and direct. For serial access, the items read or written immediately follow the previous access without concern for the underlying record structure. A pointer associated with each open file always points to the next item in the file to be accessed.

For direct access, a particular record is specified at which the access begins. In this case, the pointer is moved to the beginning of this record.

In BASIC/3000 formatted files, direct and serial access can be combined in the same file. It is possible, for instance, to position the pointer to the beginning of a record with a direct file statement, and then to access the file serially from that point.

Binary files may be accessed directly only if they are disc files with fixed length records. Otherwise, serial access must be used for binary files (Binary File Access, this section).

ASCII files may be accessed by serial or direct access. For a complete description of the restrictions on ASCII file access, see ASCII File Access in this section.

Serial File PRINT

The Serial File PRINT statement writes data items on a file, starting at the current position of the pointer. The items may be numeric or string expressions.

Form

The forms of a Serial File PRINT statement are:

PRINT #file number; print list

PRINT #file number; print list, END

PRINT #file number

PRINT #file number; END

The *print list* is a series of numeric and/or string expressions. The rules for specifying the list are the same as those described for the PRINT statement in Section II.

If the *print list* is omitted, the statement is ignored unless the file is an ASCII file in which case, a line is skipped as in a PRINT statement.

Optionally, END can be the last (or only) item in the *print list*; it writes an end-of-file mark.

Explanation

Each item in the print list is written on the file in the order it appears in the Serial File PRINT statement. The items are written starting at the position where the pointer currently appears overlaying whatever data may be in that position in the file. Record boundaries are ignored; a serial PRINT can start in the middle of one record and end in the middle of another. Each data item must, however, fit into a single record.

An embedded linefeed or a carriage return character splits a record in an ASCII file and causes them to be written out as separate records. However, these characters are not included in the printed records.

The File READ statement cannot distinguish between the different items on an ASCII file in the absence of commas, especially if the values are numeric. Therefore, when using a Serial File PRINT statement to write to an ASCII file, the data items should be separated by printing "," (or, alternatively, using the File LINPUT statement, followed by the CONVERT statement) because the Serial File PRINT statement does not write commas.

If END is the last item in the print list, an end-of-file mark is written after the last data item. When an attempt is made to read the end-of-file, an end-of-file condition occurs. If data is written immediately following the END, it overlays the end-of-file mark. If END is not specified, an end-of-record mark is written after the last data item. This is also overlaid by a subsequent PRINT.

If printing is attempted beyond the physical end of the file, an end-of-file condition occurs. The ON END statement, described in this section, specifies action to be taken when an end-of-file condition occurs. If it is not specified, the program terminates.

The file MARGIN statement (described in this section) can be used together with the PRINT # statement in order to change the length of the print line for an ASCII file.

Examples

```
  10 FILES AFILE,BFILE,*
  20 FILES AA,BB,*,*,*
  25 DIM A$[5]
  30 DIM B[2,5]
  40 MAT READ B
  50 DATA 100,200,300,400,500,600,700,800,900,1000
  60 LET A$="ABCDE"
  70 PRINT #1;"ARRAY B",(FOR I=1 TO 2,(FOR J=1 TO 5,B[I,J])),END
  80 PRINT #2;A$,B[2,1],B[1,5]
  90 PRINT #2;END
 100 PRINT #1;"END OF ARRAY"
>RUN

>DUMP AFILE
ARRAY B
 100
 200
 300
 400
 500
 600
 700
 800
 900
 1000
END OF ARRAY
>DUMP BFILE
ABCDE
 600
 500
```

The string expression "ARRAY B" followed by the entire contents of array B are written onto file number 1 (AFILE). An end-of-file mark is written following the last data item.

In line 80, the contents of the string variable A$ followed by the contents of two elements of array B are written on file number 2 (BFILE). Line 90 writes an end-of-file mark on BFILE.

Line 100 overlays the end-of-file mark previously written on AFILE with the string expression "END OF ARRAY". Since the END is omitted, an end-of-record mark is automatically written after the string expression.

Although record boundaries are ignored in a serial print, no item can be longer than a single record. The record size of AFILE was created as 106 words and BFILE as 212. Each number requires one to four words depending on type, and a string requires a word for approximately every two characters (see Appendix H, "File Structure" for exact requirements).

Serial File READ

The Serial File READ statement reads items from a file specified by file number into numeric or string variables. The first item read is the item following the current position of the pointer, that is, immediately following the last item accessed. As with serial print, record boundaries are ignored and the list of read items can start in the middle of one record and end in the middle of another.

Form

The form of a Serial File READ is:

READ # file number; read item list

The read item list is a series of variables and/or FOR loops separated by commas. The rules governing this list are the same as those described for the READ statement in Section II.

Explanation

For a formatted file each item in the specified file is read into a variable in the read item list, the first item into the first variable, the second into the second, and so forth.

The destination for a string value must be a string variable; the destination for a numeric value must be a numeric variable. Otherwise, a terminal error occurs. If the numeric value is not the same data type as the variable, conversion is performed as described in Section IV.

It is possible to check the type of the next data item with the TYP function, described later in this section.

When an attempt is made to read beyond a logical or physical end-of-file, an end-of-file condition occurs. Unless an ON END statement transfers control to another statement in the program, the program terminates.

Examples

```
 10 FILES AFILE,BFILE,*
 20 DIM A$[5],X$[10],Y$[10],C$[15]
 30 DIM B[2,5]
 40 MAT READ B
 50 DATA 100,200,300,400,500,600,700,800,900,1000
 60 LET A$="ABCDE"
 70 PRINT #1;"ARRAY B",(FOR I=1 TO 2,(FOR J=1 TO 5,B[I,J])),END
 80 PRINT #2;A$,B[2,1],B[1,5]
 90 PRINT #2;END
100 PRINT #1;"END OF ARRAY"
110 RESTORE #2
120 RESTORE #1
130 READ #1;X$,A1,B1,C1,D1,E1
140 PRINT X$,LIN(1),A1,B1,C1,D1,E1
150 READ #1;A2,B2,C2,D2,E2
160 PRINT A2,B2,C2,D2,E2
170 READ #2;Y$,A,B
180 PRINT Y$,A,B
190 READ #1;C$
195 PRINT C$
200 READ #1;X
210 REM..ATTEMPT TO READ END-OF-FILE CAUSES TERMINATION
>RUN

ARRAY B
 100           200           300           400           500
 600           700           800           900           1000
ABCDE          600           500
END OF ARRAY
END OF FILE IN LINE 200
```

After data is written on files 1 and 2 with the print statements in lines 70-100, and the pointer is restored to the start of each file in lines 110 and 120, the data that was written can be read.

The first six items in file 1 are read in line 130. The next five items are read in line 150. The three items written on file 2 are read in line 170. PRINT statements are inserted to test the accuracy of the reads and the previous writes.

A string item remains in file number 1; this is read in line 190. Line 200 attempts to read an end-of-file causing the message: END OF FILE IN LINE 200 to be printed.

When a string is read from a binary file, the number of characters read depends on the form of the variable. For instance, if A$ is a simple string variable:

READ #1;A$	reads the physical length of A$
READ #1,;A$(I)	reads the physical length of the substring starting at I
READ #1;A$(I,J)	reads J–I+1 characters into the substring starting at I
READ #1;A$(I;J)	reads J characters into the substring starting at I

File RESTORE Statement

The File RESTORE statement repositions the file pointer to the start of the file. The statement can be used for any file, but is particularly useful for serial files such as magnetic tape.

Form

RESTORE # file number

The *file number* identifies a file that is currently open.

Explanation

When File RESTORE is executed, the file pointer is set to point to the beginning of the first record in the file. A serial read or print will begin at that position.

Example

```
 10 FILES AFILE,BFILE
 20 PRINT #1,1;123.4
 30 PRINT #2,1;567.8
 40 RESTORE #2
 50 RESTORE #1
 60 READ #1;C
 70 READ #2;D
 80 PRINT C,D
>RUN
 123.4           567.8
```

When the File RESTORE statements are executed, the pointer in file number 2 is moved back to the start of that file. Then the pointer in file number 1 is moved to the start of that file. If the files are magnetic tape, they are rewound.

Direct File PRINT

The Direct File PRINT statement writes a list of data items onto the specified file as a single record. Printing begins at a particular record specified in the PRINT statement. After printing, an end-of-record mark is written and any data previously contained in the record is lost. Data in records preceding and following the specified record is not changed.

Form

The forms of a Direct File PRINT are:

PRINT # file number, record number; print list

PRINT # file number, record number; print list, END

PRINT # file number, record number

PRINT # file number, record number; END

Both the *file number* and *record number* are integer expressions. The *print list* is optional and has the same format as a Serial File PRINT. If it is missing, the statement erases the contents of the specified record.

Explanation

The Direct File PRINT positions the pointer at the beginning of the specified record and then writes the contents of the print list. An end-of-record mark is written following the items in the print list. Any previous end-of-record marks are ignored.

When a string item is printed to a binary file, the item starts at a word boundary and is as long as the current length of the string. If the string is of odd length, an additional character of undefined value is written to complete the last word. Each word contains two characters.

The first record of the file is record number 1.

END writes an end-of-file mark. If no print list is specified, any data in the specified record is replaced by the end-of-file mark.

Serial and Direct PRINT statements can be used to write on the same file. A serial print following a direct print will write its data items immediately following the previous items.

The file MARGIN statement (described in this section) can be used together with the PRINT # statement in order to change the length of the print line for an ASCII file.

Examples

```
 10 FILES AFILE,BFILE,*
 20 FILES AA,BB,*,*,*
 30 ASSIGN "XFILE",3,X1,NL
 40 ASSIGN "DFILE",6,D1,RR
 50 LET A1=1,B1=2,C1=3,D1=4,E1=5
 60 LET A$="A"
 70 FOR N=1 TO 10
 80   LET B[N]=N+1
 90 NEXT N
100 DIM B[10]
110 PRINT #3,1;"START OF XFILE"
120 PRINT #3,2;10,A1,(FOR N=1 TO 10,B[N])
130 REM..TWO RECORDS HAVE BEEN WRITTEN ON XFILE
140 PRINT #6,2;B1,C1,D1,E1
150 PRINT #6,1;A$
160 PRINT #6,3;END
170 REM..THE THIRD RECORD OF DFILE IS AN END-OF-FILE
>RUN
```

The first record of file number 3 contains a string value. The second record has two numeric items and the contents of a 10-element numeric array. No end of file is written on #3.

File number 6 also has a string item in its first record; it has four numeric items in the second record. The third record is an end-of-file.

Note that the records do not have to be written in the order they appear in the file.

Direct File READ

The Direct File READ statement reads data values starting at a specified record of a specified file and assigns them to variables. Numeric values can be assigned only to numeric variables, and string values to string variables, as in Serial File READ.

Form

The forms of the Direct File READ statement are:

> *READ # file number, record number; read item list*
>
> *READ # file number, record number*

The *file number* and *record number* are integer expressions. The optional *read item list* is of the same form as in a READ statement.

Explanation

Data values are read from the record and assigned to the variables in the item list. If a record number is specified outside the range of the file, an end-of-file condition occurs.

If the *read item list* is omitted, the statement moves the file pointer to the beginning of the specified record, but does not read any data.

A READ statement for an ASCII file is terminated by a NULL character; that is, NUM (character)=0. Therefore, binary data should not be written to an ASCII file.

Example

```
  10 DIM C[2,5]
  20 DIM A$[20],X$[20]
  30 READ #3,1;A$
  40 READ #3,2;X,Y,(FOR N=1 TO 2,(FOR P=1 TO 5,C[N,P]))
  50 READ #6,1;X$
  60 PRINT A$,X,Y
  70 MAT PRINT C
  80 PRINT X$
  90 FILES *,*,XFILE,*,*,DFILE
>RUN

START OF XFILE  10                   1
 2                3                  4                  5                  6

 7                8                  9                  10                 11
A
```

In this example, the data previously written on records 1 and 2 of XFILE and on record 1 of DFILE are read. (See examples with Direct File PRINT.)

In the example below, ten records are written on file AA and then these records are copied to file BB and AA is closed.

```
 10 DIM X[10]
 20 MAT READ X
 30 ASSIGN "AA",1,A
 40 FOR R=1 TO 10
 50   PRINT #1,R;X[R]
 60 NEXT R
 70 PRINT #1;END
 80 ASSIGN "BB",2,B
 90 FOR R=1 TO 10
100   READ #1,R;X
110   PRINT #2,R;X
120 NEXT R
130 PRINT #2;END
140 PRINT "FILE AA COPIED TO FILE BB"
150 ASSIGN *,1
160 DATA 10,20,30,40,50,60,70,80,90,100
170 FILES *,*
>RUN
FILE AA COPIED TO FILE BB
```

ASCII File Access

ASCII files may be accessed with any statements except the ADVANCE, UPDATE or CREATE statements. The File LINPUT statement can be used to read the contents of an ASCII record and the File MARGIN statement can be used to change the length of the print line for an ASCII file. In addition, the PRINT # USING and MAT PRINT # USING statements (see Section IX) allow the user to print to an ASCII file according to a specified format.

File LINPUT Statement

The File LINPUT statement reads the entire contents of a record in an ASCII file into a string variable. File LINPUT is used to read ASCII files only.

Form

LINPUT # file number; string variable
LINPUT # file number, record number; string variable

Explanation

File LINPUT reads the contents of the record at which the pointer is currently positioned or at the specified record. This is like LINPUT (see Section II) except that input is from a file, not a terminal, and a record, not a line, is read.

If the string variable is not large enough to contain the entire record, the extra characters are discarded.

Example

```
  10 DIM A$[72],B$[20]
  20 READ A1,B1,C1,D1,E1
  30 RESTORE #10
  40 PRINT #10;A1,B1,C1,D1,E1
  50 PRINT #10;"1, JANUARY,1973"
  60 RESTORE #10
  70 LINPUT #10;A$
  80 LINPUT #10;B$
  90 PRINT A$,B$
 100 DATA 10,20,30,40,50
 110 FILES AFILE,BFILE,*,AA,BB,DFILE,FF2,X,*,ASCI
>RUN
 10               20               30               40               50
1, JANUARY,1973
```

The first two records of the ASCII file ASCI are input into the string variables A$ and B$, the first record in A$ and the second in B$.

File MARGIN Statement

The File MARGIN statement is used to set the length of the print line (number of characters) for the PRINT and PRINT # statements. File MARGIN is used for ASCII files only.

Form

MARGIN *marginsize*
MARGIN #*file number,marginsize*

Explanation

The optional *file number* identifies an ASCII file. If the *file number* is zero or is not specified, the length of the print line for the PRINT and MAT PRINT statements is set. Otherwise, the length of the print line is set for the specified ASCII file; any PRINT # or MAT PRINT # statement to the same file which follows the MARGIN statement will be affected. The length of the print line remains set until the next MARGIN statement is encountered or until the program terminates.

The *marginsize* gives the desired length of the print line. The value specified is rounded to the nearest integer, which may not be less than 15 or greater than the record size. If a *marginsize* outside these bounds is specified, a warning message is given and the *marginsize* will be set as follows: if the rounded off value is less than 15, the *marginsize* is set to 15, while if it is greater than the record size, it is set to the record size.

Both the *file number* and *marginsize* may be numeric constants, variables, or expressions.

The MARGIN statement has no effect on the PRINT USING, PRINT # USING, MAT PRINT USING, and MAT PRINT # USING statement described in Section IX.

Example

BFILE is an ASCII file of record length 40. If a marginsize greater than 40 or less than 15 is specified, a warning is given and the marginsize is set to 15 or to 40 as appropriate.

```
   10 FILES BFILE
   20 FOR I=1 TO 4
   30   PRINT "MARGIN SIZE";
   40   INPUT J
   50   MARGIN #1,J
   60   PRINT #1;1,2,3,4,5
   70   PRINT #1
   80 NEXT I
>RUN
MARGIN SIZE?45

WARNING: ILLEGAL MARGIN IN LINE 50
MARGIN SIZE?40
MARGIN SIZE?20
MARGIN SIZE?10

WARNING: ILLEGAL MARGIN IN LINE 50
```

BFILE now contains the following:

```
1             2             3
4             5

1             2             3
4             5

1             2
3             4
5

1
2
3
4
5
```

Binary File Access

Access to binary files may be serial or direct as with BASIC formatted files. Direct access is allowed only for files on disc with fixed length records. Serial access is allowed for all files.

Data items are stored in binary files as binary words without type flags. When data is read from a binary file, it is assumed to be the same type as the variable into which it is being read. Items in binary files can cross record boundaries and new records are used only upon overflow.

Dynamic Locking

If a file is opened with an ASSIGN statement and either WL or NL is specified as a restriction, access to the file can be dynamically controlled with the LOCK and UNLOCK statements.

Form

LOCK # file number

UNLOCK # file number

The file identified by *file number* must have been opened with an ASSIGN statement specifying one of the restrictions WL or NL.

Explanation

LOCK gives a program exclusive control of a file until it is unlocked by the UNLOCK statement. During control by LOCK, no other program can lock the file until UNLOCK is executed. An attempt to lock a file that has been locked by another program will cause the program to be suspended until the file has been unlocked in the other program. Only one file at a time can be locked, although WL or NL may be specified for more than one file. Any write operations on a locked file are guaranteed to be physically completed before the UNLOCK is executed. Each access to a file during dynamic locking should be made between a LOCK and UNLOCK statement.

Note that the LOCK statement does not actually restrict other programs from accessing the file. Therefore, all programs must cooperate by first locking, then accessing, and then unlocking the file. Dynamic locking is not necessary if it is unlikely that more than one user will access the same file, or if none of the users are writing on the file.

Examples

```
  10 FILES AFILE,BFILE,*
  20 FILES AA,BB,*,*,*
  30 ASSIGN "X",8,T,NL
  40 PRINT T
  50 LET A=2.57325E13
  60 LOCK #8
  70 PRINT #8;A
  80 UNLOCK #8
>RUN
LOCK
 Ø
```

The file is locked in line 60. This assures that no other program can lock the same file, and that the write operation in line 70 will be completed and the pointer moved to the beginning of the next record. Line 70 contains a Serial File PRINT statement that writes the contents of the variable A onto file number 8. Line 40 prints the contents of the numeric variable T containing the result of the ASSIGN. This shows that the file is open and can be written on before LOCK is specified. Following execution of the UNLOCK statement, in line 80, any other user may lock the file.

If a program performs multiple file locking, it must be run with MR status. There are two ways to achieve MR status:

- The first method uses the MR parameter of the RUN command. The user must have MR capability to execute the program with its MR status.
- The second method uses the SAVE command with the MR parameter. To perform the SAVE, the user must have MR capability as in the previous case. (The SAVE command gives MR status only to the saved program, not the current program.)

If the user does not possess MR capability, a saved program with MR status can still be retrieved and run with its MR status by either the RUN or the GET command. However, if the program is modified, it will lose its MR status. In this case, MR status can be regained by one of the two methods described above.

ON END Statement

The ON END statement sets a flag for a specified file so that if and when an end-of-file condition occurs in reading and writing that file, control is transferred to a specified statement. If the flag is not set, an end-of-file condition causes program termination.

Form

The forms of ON END are

ON END # file number THEN label

IF END # file number THEN label

ON END and IF END are accepted interchangeably but the statement is always stored internally and listed as ON END.

Explanation

When an end-of-file condition occurs during execution of a Direct or Serial File READ statement or a File LINPUT statement, the ON END statement transfers control to the statement identified by *label.* When writing on a file with a Direct or Serial File PRINT statement, ON END transfers control to *label* when an attempt is made to write past the physical end-of-file.

ON END is an executable statement and the transfer label can be altered by a subsequent ON END statement. The label must not lie within the range of a function unless the ON END is also within that function. An ON END within a function must refer to a label within that function; ON END has no effect outside the function.

ON END is not executed if an end-of-file is encountered during execution of the ADVANCE statement (see ADVANCE description, this section).

Examples

In the example below, the Direct File READ in line 70 attempts to read an end-of-file written in line 50. The ON END statement transfers control to line 100.

```
  10 FILES AFILE,BFILE
  20 READ A1,B1,C1,D1,E1
  30 DATA 100,200,300,400,500
  35 ON END #2 THEN 100
  40 PRINT #2,1;A1,B1,C1
  50 PRINT #2,2;D1,E1,END
  60 READ #2,1;A,B,C
  70 READ #2,2;D,E,F
  80 END
 100 PRINT "END OF FILE 2"
 120 PRINT A,B,C,D,E
>RUN
END OF FILE 2
 100            200            300            400            500
```

In the next example, file RR is created with 5 records. The File PRINT statement in line 100 attempts to write past the end-of-file and the ON END statement causes a transfer to line 130.

```
>CREATE RR,5
  10 FILES RR
  20 DIM X[6]
  30 MAT READ X
  40 ON END #1 THEN 130
  50 PRINT #1,1;X[1]
  60 PRINT #1,2;X[2]
  70 PRINT #1,3;X[3]
  80 PRINT #1,4;X[4]
  90 PRINT #1,5;X[5]
 100 PRINT #1,6;X[6]
 110 DATA 10,20,30,40,50,60
 120 END
 130 PRINT "END OF FILE"
>RUN
END OF FILE

>DUMP RR
 10
 20
 30
 40
 50
```

ADVANCE Statement

The ADVANCE statement allows for skipping past items in a BASIC/3000 formatted file without reading them.

Form

The form of ADVANCE is

ADVANCE # file number; integer expression, numeric variable

The *integer expression* specifies the number of data items to be skipped and the *numeric variable* is used to return a result value.

Explanation

If the integer expression is negative, items are skipped in a reverse direction.

After execution of ADVANCE, the numeric variable equals zero if the ADVANCE was successful. If the ADVANCE statement encountered either an end-of-file or a start-of-file, the numeric variable equals the difference between the number of items requested and the number actually skipped. This value is negative if ADVANCE was in the reverse direction.

Example

```
  10 FILES *,*,AA,BB
  20 LET A=1,B=2,C=3,D=4,E=5,X$="X"
  30 PRINT #3;A,B,C,D,E,X$,END
  40 RESTORE #3
  50 ADVANCE #3;3,X1
  60 IF X1<>0 THEN GOTO 100
  70 READ #3;L,M
  80 READ #3;A$
  90 PRINT L,M,A$
 100 PRINT X1
>RUN
 4              5              X
 0
```

The first three items in file AA are skipped, then the next three are read and printed.

UPDATE Statement

The UPDATE statement allows an item in a BASIC/3000 formatted file to be modified without affecting any of the items that precede or follow. UPDATE overwrites the next item in the file and positions the pointer to the item that follows.

Form

The form of UPDATE is

UPDATE # file number; expression

Explanation

If the existing data item is numeric, the *expression* to be written must be numeric also. If their types do not match, the value of the expression is converted to the type of the existing data item.

If the existing item is a string, the expression must be a string. The string expression is truncated or blank-filled on the right to fit the size of the existing item exactly.

Examples

```
  10 FILES *,*,AA
  20 LET A=1,B=2,C=3,D=4,E=5
  30 LET A$="A",B$="B"
  40 PRINT #3,1;A,B,C,D,E,A$,B$
  50 RESTORE #3
  60 ADVANCE #3;3,X1
  70 UPDATE #3;4.57
  80 ADVANCE #3;2,X2
  90 UPDATE #3;"XYZ"
>RUN

>DUMP AA
 1
 2
 3
 4.57
 5
A
X
```

The fourth and seventh items in file AA are given new values with the UPDATE statement. The file is then dumped to illustrate the successful update.

Listing File Contents

DUMP COMMAND

The DUMP command displays the contents of a BASIC/3000 formatted file on another file: either the normal output file or a specified ASCII file. DUMP provides a simple way to print file contents at the terminal. Normally the contents of a file to be dumped are string data.

Form

DUMP file name

DUMP file name, OUT=asciifile

file name is a simple string without quotes. It must name a BASIC/3000 formatted file. *asciifile* is the name of an existing ASCII file.

Explanation

The contents of the named file are dumped on the normal output file (the terminal) unless *OUT=asciifile* is specified, in which case, the file is dumped on the specified ASCII file.

DUMP prints each item on a separate line; record boundaries are not indicated. DUMP terminates at the first end-of-file.

If the length of a string is greater than that of the output file records, the string is truncated. In this case, MPE file commands are used to change the record length during a BREAK.

```
:FILE outfile=$STDLIST;REC=-256
:BASIC
HP32101B.00.08(4WD)   BASIC   (C)HEWLETT-PACKARD CO 1976
>CREATE NFILE,50
>100 FILES NFILE
>200 PRINT #1,1;40,400
>300 PRINT #1,2;50,500,5000
>400 PRINT #1,3;"HANDYMAN"
>RUN
>DUMP NFILE,OUT=outfile
 40
 400
 50
 500
 5000
HANDYMAN

>EXIT

 END OF SUBSYSTEM
```

Example

```
 10 FILES AFILE
 20 LET X=10.5,Y=75,Z=150
 30 PRINT #1,1;10,20,30
 40 PRINT #1,2;"HELLO"
 50 PRINT #1,3;X,Y,Z
>RUN

>DUMP AFILE
 10
 20
 30
HELLO
 10.5
 75
 150
```

File Functions

Three functions are available in BASIC/3000 that assist in file access. They are TYP, REC, and ITM.

TYP FUNCTION

The TYP function returns the type of the next data item for a particular file. This function is used in conjunction with File READ statements since the variables into which the data is read must be string if the item is string, numeric if the item is numeric.

Form

TYP (file number)

The file number may be a numeric constant, variable, or expression, and must identify an existing file.

Explanation

The value returned by TYP depends on the type of the next data item in the file.

TYP(x)	Meaning
1	real
2	string
3	end-of-file
4	end-of-record
5	integer
6	long
7	complex

If the *file number* is greater than zero, the file is treated as a serial file. This means that the value 4 is never returned as end-of-record marks are skipped in a serial file.

If the *file number* is less than zero, the file is treated as a direct file and any of the values of TYP may be returned. Since the file number is based on the absolute value of the expression, an expression equal to –1 means that file number 1 is examined and treated as a direct file.

If the *file number* equals zero, TYP returns a result based on the current position of the pointer to the DATA statements (see READ/DATA/RESTORE description, Section II.) The value 4 is never returned, and 3 means end-of-data.

If the file is binary, TYP returns only the vaues 1, 3, or 4.

If the file is ASCII, TYP returns the same results with the same meaning as the BUF function, except that TYP returns a value of 3 for end-of-file and a value of 4 for end-of-record. (See BUF Function description under INPUT Statement, Section II.)

Examples

File AFILE is written with integer, real, long, and complex numbers. A later program reads only the long and complex items using the TYP function to distinguish them:

```
  10 FILES AFILE
  20 INTEGER A,B
  30 LONG L1
  40 COMPLEX C1
  50 LET A=25,B=100,X=2.357E12,Y=7.649E-5
  60 LET L1=9.99999L-5
  70 LET C1=(.002359,.002175)
  80 PRINT #1;A,X,L1,B,Y,C1,"STRING RECORD",END
>RUN

>SCRATCH

  10 FILES AFILE
  20 LONG L3
  30 COMPLEX F1
  40 GOTO TYP(1) OF 50,50,130,50,50,70,100
  50 ADVANCE #1;1,A1
  60 GOTO 40
  70 READ #1;L3
  80 PRINT L3;
  90 GOTO 40
 100 READ #1;F1
 110 PRINT F1;
 120 GOTO 40
 130 PRINT LIN(1);"END OF FILE"
>RUN
 9.999990000000000L-05   ( 2.35900E-03, 2.17500E-03)
END OF FILE
```

REC FUNCTION

The REC function returns the record number at which the file pointer is currently positioned for a specified file.

Form

REC(file number)

The file number identifies an existing file. REC returns the record number of the record in that file currently being accessed.

Example

```
 10 FILES AFILE,BFILE,XFILE,AA,BB
 20 LET A=1,B=2,C=3,D=4,E=5
 30 PRINT #5;A,B,C,D,E
 40 LET R=REC(5)
 50 READ #5,R; V,W,X,Y,Z
 60 PRINT V,W,X,Y,Z
>RUN
 1                  2                  3                  4                  5
```

The variable R is set to the value of REC(5); this vaue, the current record in file BB, is then used in the Direct File READ in line 50.

ITM FUNCTION

The ITM function returns the number of data items between the beginning of the currently accessed record and the position of the file pointer for a specified file.

Form

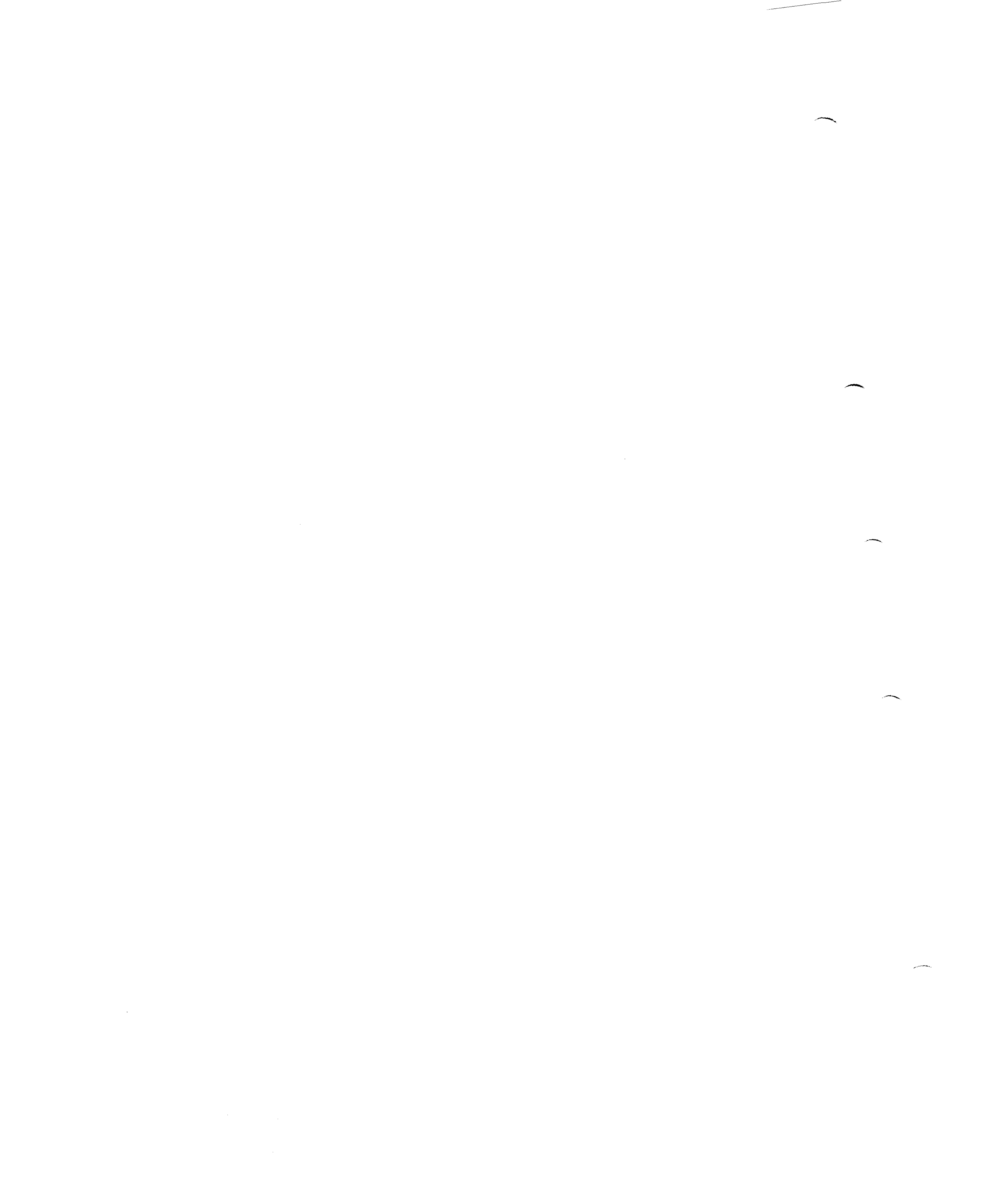
ITM*(file number)*

The *file number* identifies a currently open BASIC formatted file. ITM returns a real value showing the number of data items from the beginning of the current record through the position of the file pointer for that file.

Example

```
  10 FILES TEXT
  20 PRINT #1;12345,23456,34567
  30 RESTORE #1
  40 READ #1;A,B
  50 PRINT A,B
  60 PRINT "ITEM";ITM(1);"HAS JUST BEEN READ ";
  65 PRINT "FROM RECORD";REC(1)
  70 PRINT #1,2;"ABCDEF"
  80 DIM A$[6]
  90 READ #1,2;A$
 100 PRINT A$
 110 PRINT "ITEM";ITM(1);"HAS JUST BEEN READ ";
 115 PRINT "FROM RECORD";REC(1)
>RUN
  12345          23456
 ITEM 2     HAS JUST BEEN READ FROM RECORD 1

 ABCDEF
 ITEM 1     HAS JUST BEEN READ FROM RECORD 2
```



File Array Operations

There are four statements for accessing files with arrays:

Serial File MAT PRINT
Serial File MAT READ
Direct File MAT PRINT
Direct File MAT READ

For formatted output of arrays, see the descriptions of MAT PRINT USING and MAT PRINT # USING in Section IX.

SERIAL FILE MAT PRINT STATEMENT

This statement prints entire arrays on a file starting at the current position.

Form

MAT PRINT #file number; array list

MAT PRINT #file number; array list, END

MAT PRINT #file number; END

The *array list* contains a list of array names separated by commas. END writes an end-of-file mark.

Explanation

The arrays specified in the array list are written on the specified file row by row. END, whether alone or after the array list, writes an end-of-file.

The length of the print line can be changed by using the MARGIN statement described in this section.

SERIAL FILE MAT READ STATEMENT

This statement reads data items, starting with the current position of a specified file, to fill entire arrays row by row.

Form

MAT READ #file number; array list

Explanation

The array names in the array list will contain data read from the specified file.

Example

The example below writes the values of two arrays onto the file BB; it then reads the array value into two different arrays with different dimensions:

```
 10 FILES *,*,*,BB
 20 DIM A[2,5],B[3,2]
 30 MAT READ A
 40 MAT READ B
 50 DATA 10,20,30,40,50,60,70,80,90,100,1,2,3,4,5,6
 60 MAT PRINT #4;A,B
 70 RESTORE #4
 80 DIM C[5,2],D[2,3]
 90 MAT READ #4;C,D
100 MAT PRINT C,D
>RUN
 10                 20

 30                 40

 50                 60

 70                 80

 90                 100

 1                  2                   3

 4                  5                   6
```

DIRECT FILE MAT PRINT STATEMENT

This statement prints arrays starting at the beginning of a specified record within a specified file.

Form

MAT PRINT # file number, record number; array list

MAT PRINT # file number, record number; array list, END

MAT PRINT # file number, record number; END

Explanation

The array names in the list are written on the file starting at the beginning of the record identified by record number. An END in the array list causes an end-of-file mark to be written. The contents of the array list may cross record boundaries.

The length of the print line can be changed by using the MARGIN statement described in this section.

DIRECT FILE MAT READ STATEMENT

This statement reads entire arrays starting from a specified record in a specified file. The read may cross record boundaries.

Form

MAT READ # file number, record number; array list

The *array list* contains the names of entire arrays.

Explanation

The contents of the specified record number are read into the array variables specified in the array list.

Examples

Two arrays A and B are written on MAT1 starting at record number 5. The arrays are read from record number 5 into arrays C and D respectively.

```
 10 FILES *,*,*,*
 20 DIM A[10],B[3,2],C[5,2],D[2,3]
 30 MAT READ A,B
 40 DATA 1,2,3,4,5,6,7,8,9,0
 50 DATA 6.5,7.4,8.3,9.2,.1,5.5
 60 ASSIGN "MAT1",4,M1
 70 MAT PRINT #4,5;A,B
 80 MAT READ #4,5;C,D
 90 MAT PRINT C,D
>RUN
 1                  2

 3                  4

 5                  6

 7                  8

 9                  0

 6.5                7.4                8.3

 9.2                .1                 5.5
```