Time-Sharing Service

Reference Manual

Information Systems

Time-Sharing Service

BASIC Language Extensions

4/68

GENERAL ELECTRIC

INFORMATION SERVICE DEPARTMENT

Time-Sharing Service

BASIC LANGUAGE EXTENSIONS

Reference Manual

February 1968
Reprinted 4-68

GENERAL ELECTRIC

INFORMATION SERVICE DEPARTMENT

802207

PREFACE

This manual, which replaces publications no. 708206 and 710215A, is an addendum to the BASIC* Language Reference Manual (202026) describing a language for the General Electric Time-Sharing System. This manual should be used with the BASIC Language Reference Manual. When the latter manual is revised, information about the extensions to BASIC will be included in it, and this manual will become obsolete.

Section 2 of this addendum should be used for information about matrix operations, since it contains more descriptive information about the function.

* Developed by Dartmouth College

CONTENTS

1. UNARY MINUS

In the previous version of BASIC the unary minus was given the highest priority of execution among the arithmetic operators. The priority order of execution was:

(1) unary minus
(2) power
(3) division and multiplication
(4) addition and subtraction

In the extensions to BASIC, the high priority for the unary minus has been eliminated. All negations are considered as the subtraction operator. The following comparative examples will illustrate the point.

Original BASIC	Extensions to BASIC
LET X = -2	LET X = -2
LET Y = -X ↑ 2	LET Y = -X↑ 2
yielded Y = 4 i.e. Y = (-X)↑ 2	yields Y = -4 i.e. Y = -(X ↑2)

This change is a result of user suggestions and requests, and reflects the consensus of time-sharing users. If you have programs that use the unary minus, be sure to modify those lines to reflect the new priority.

2. MATRIX OPERATIONS*

The matrix operation statements available in BASIC and the extensions to BASIC are among the most powerful and useful in the entire language.

The following is a list of available matrix commands. Use of each of the commands is described in detail in Sections 2.1 to 2.8. Examples in Section 2.9 illustrate their use.

MAT READ A,B,C	Read the matrices, A,B,C, their dimensions having been previously specified. Data is read in row-wise sequence.
MAT PRINT A,B;C	Print the matrices A,B,C, with A and C in the regular format, but B closely packed.
MAT C = A + B	Add the two matrices A and B and store the result in matrix C.
MAT C = A - B	Subtract the matrix B from the matrix A and store the result in matrix C.
MAT C = A * B	Multiply the matrix A by the matrix B and store the result in matrix C.
MAT C = INV(A)	Invert the matrix A and store resulting matrix in C.
MAT C = TRN(A)	Transpose the matrix A and store the resulting matrix in C.
MAT C = (K) * A	Multiply the matrix A by the value represented by K. K may be either a number or an expression, but in either case it must be enclosed in parentheses.

*The only changes resulting from extensions to BASIC involve dimensioning (Section 2.8). The other matrix operations are not new, but this section is more descriptive than the original BASIC manual version.

MAT C = CON	Set each element of matrix C to one. (CON = constant.)
MAT C = ZER	Set each element of matrix C to zero.
MAT C = IDN	Set the diagonal elements of matrix C to one's and the non-diagonal elements to zeroes, yielding an identity matrix.

2.1 MAT READ AND MAT PRINT

Data may be read into or printed from a matrix without having to reference each element of the matrix individually by using the MAT READ and MAT PRINT commands.

Examples:

```
100 MAT READ A,F,H,G

150 MAT PRINT C

175 MAT READ Z

190 MAT PRINT A,L
```

Information is read into a matrix using the DATA statement. The elements in the DATA statement are taken in row order (i.e., $A_{1,1}, A_{1,2}, \ldots, A_{1,m}, A_{2,1}, A_{2,2}, \ldots, A_{2,m}, \ldots, A_{n,m}$).

Information is read from DATA statements until the matrix array is completely filled. Partial matrices may not be read or printed.

Example:

```
110 DIM L(2,3), M(2,2)

150 MAT READ L,M

160 LET L(2,2) = -2*L(2,2)

200 MAT PRINT L,M

500 DATA 1,2,3,4,5,6,3,-12,0,7
```

L is defined as a 2 by 3 matrix and M as a 2 by 2 matrix (line number 110). The MAT READ statement reads from the DATA statement located at line number 500 in row order. The matrix element, $L_{2,2}$ is recomputed at line number 160. The two matrices are then printed to yield:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & -10 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 3 & -12 \\ 0 & 7 \end{bmatrix}$$

2.2 MATRIX ADDITION, SUBTRACTION, AND MULTIPLICATION

Matrices can be added, subtracted and multiplied using the matrix arithmetic commands. The matrix dimensions must be conformable for each operation. If dimensions are not conformable, execution is terminated and you receive a dimension error message.

The matrix arithmetic statements may take three forms. They are:

```
MAT C = A+B
                    or          MAT P = Q*R
MAT J = G-P
```

Only one operation may be performed per statement.

Example:

Calculate [H] a,a = [E] a,a - [K] a,a + [A] a,3 * [B] 3,a

```
612 MAT H=A*B
615 MAT H=H+E
618 MAT H=H-K
```

2.3 SCALAR MULTIPLICATION

A matrix can be multiplied by a scalar expression using the command:

MAT X=(expression) *D

where X and D are matrices and the expression in parentheses is a scalar quantity. The parentheses are required to indicate scalar multiplication rather than matrix multiplication. Only one operation may be performed per statement.

Examples:

```
10 MAT F=(2)*G
50 MAT Q=(2.33+M)*Q
75 MAT B=(N)*A
```

2.4 IDENTITY MATRIX

An identity matrix is defined by the statement:

MAT B=IDN

or MAT R=IDN (expression, expression)

In the first statement, the matrix variable B is set up as an identity matrix. If B is not defined to be square, you will receive a dimension error message. In the second statement, the size of the identity matrix R is determined at execution time by the value of the expression enclosed in parentheses.

Examples:

```
 90 MAT A=IDN
100 MAT V=IDN(2*N+1,2*N+1)
120 MAT B=IDN(Q,Q)
130 MAT W=IDN
140 MAT C=IDN(1,1)
```

2.5 MATRIX TRANSPOSITION

Matrices are transposed using the form:

MAT Y=TRN(Z)

where Y and Z are both matrices. The matrix Z transpose will replace matrix Y. Y and Z must conform for transposition.

Examples:

300 MAT G=TRN(H)

400 MAT U=TRN(V)

Matrix transposition in place (MAT A=TRN(A)) is not allowed.

2.6 MATRIX INVERSION

Matrices are inverted using the form:

MAT I=INV(J)

where I and J are both matrices. I will contain the matrix J inverse. I and J must conform for inversion.

Examples:

550 MAT K=INV(L)

560 MAT A=INV(B)

Matrix inversion in place (MAT A=INV(A)) is not allowed. If a matrix is singular, you will receive the message NEARLY SINGULAR MATRIX.

2.7 MATRIX ZER AND CON FUNCTIONS

The ZER function is used to zero out all elements of a matrix. It may also be used to redefine the dimensions of a matrix during execution as described in 2.8. As an example

MAT C=ZER

will zero out the elements of matrix C.

The CONstant function is used to set all elements of a matrix to one. As an example

MAT C=CON

will set all elements of matrix C to one.

2.8 DIMENSIONING

Every matrix variable used in a program must be given a single-letter name.

A matrix variable must be defined in a DIM statement. This sets aside the amount of storage required by the matrix variable during execution of the program. For example:

DIM P(3,4),Q(5,5)

The DIM statement defines both a P and Q matrix. P is defined as a 12 element matrix, and Q is defined as a 25 element matrix. Note that the first element of P is P(1,1) and the last element P(3,4). The elements of Q run from Q(1,1) through Q(5,5). All matrix variables must be doubly dimensioned as done here.

Prior to any computation using the MAT statements, you must declare the precise dimensions of all matrices to be used in the computation. Four of the MAT statements themselves are used to accomplish this dimensioning. They are:

```
MAT READ C(M,N)

MAT C = ZER(M,N)

MAT C = CON(M,N)

MAT C = IDN(N,N)
```

The first three statements specify matrix C as consisting of M rows and N columns. The fourth statement specifies matrix C as a square matrix of N rows and N columns. These same instructions may be used to redimension a matrix during running. A matrix may be redimensioned to either a larger or a smaller matrix provided the new dimensions do not require more storage space than was originally reserved by the DIM statement. To illustrate, consider the following statements:

```
 10 DIM A(8,8),B(8,8),C(8,8)

 50 MAT READ A(2,2),B(2,2)

 60 MAT C = ZER(2,2)

              .
              .
              .

100 MAT A = IDN(8,8)

110 MAT READ B(4,4),C(4,4)
```

Observe that the DIM statement reserves enough storage to accommodate three matrices, each consisting of 64 elements. The initial MAT READ specifies the dimensions of both matrices A and B as two rows and two columns.

The MAT READ also reads the number of values required by the dimensions into the storage which was reserved by the DIM statement. The MAT READ reads the values in row-wise sequence. In the initial MAT READ, the elements in the order read are A(1,1), A(1,2), A(2,1), A(2,2), B(1,1), B(1,2), B(2,1), and B(2,2). Statement 60 illustrates the ZER being used to specify dimensions and to zero the elements of the matrix C. Statements 100 and 110 illustrate the redimensioning concept, where matrix A is redimensioned as an eight row, eight column identity matrix and matrices B and C are redimensioned as four row, four column matrices into which data is to be read.

While the combination of ordinary BASIC instructions and MAT instructions makes the language very powerful, you must be careful about your dimensions. In addition to having both a DIM statement and a declaration of current dimension, you must be careful with the eleven MAT statements. For example, a matrix product MAT C = A*B may be illegal for one of two reasons: A and B may have dimensions such that the product is not defined, or C may have the wrong dimensions for the answer. In either case you will receive the DIMENSION ERROR message.

2.9 EXAMPLES

Two programs follow which illustrate some of the capabilities of the MAT instructions. In the first example, the values for M and N are read. Using these values as indices, statement 30 sets the dimensions for matrices A, B, D, and G respectively. Also the values for the elements of these matrices

are read. In sequence then, the dimensions of matrix C are specified and the elements set to zero (statement 40), matrix A is printed (statement 60), matrix B is printed (statement 80), the sum of matrices A and B is found and stored in C (statement 90), matrix C is printed (statement 110), the dimensions for matrix F (a vector) are set and the elements set to zero (statement 120), the product of matrices C and D is computed and stored in F (statement 130), the dimensions for matrix H (single value) are specified and the elements set to zero (statement 140), and finally the product of matrices G and F is found and stored in H and printed (statements 150,170).

In the second example, a value N is read which determines the order of the Hilbert matrix segment to be computed, stored, and printed. Next this matrix is inverted and printed. Finally the Hilbert matrix is multiplied by its own inverse and the resulting product matrix is printed. Notice that in example 2 line 190 specifies N=2 to produce the first 3 matrices of order 2, later returns to read in the data "3", redimensions to a larger array (larger than 2, but less than the original 20) and produces more output.

MATRIX PROGRAM EXAMPLE 1

```
10 DIM A(5,5),B(5,5),C(5,5),D(5,5),E(5,5),F(5,5),G(5,5),H(5,5)
20 READ M,N
30 MAT READ A(M,M),B(M,M),D(M,N),G(N,M)
40 MAT C=ZER(M,M)
50 PRINT "MATRIX A ØF ØRDER "M
60 MAT PRINT A;
70 PRINT "MATRIX B ØF ØRDER "M
80 MAT PRINT B;
90 MAT C= A+B
100 PRINT "       C=A+B"
110 MAT PRINT C;
120 MAT F=ZER(M,N)
130 MAT F=C*D
140 MAT H=ZER(N,N)
150 MAT H=G*F
160 PRINT "  H  "
170 MAT PRINT H;
1800 DATA 3,1
1810 DATA 1,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,1,1,2,3,3,2,1
1999 END

RUN

BASICT

MATRIX A ØF ØRDER  3
 1      2      3

 4      5      6

 7      8      9

MATRIX B ØF ØRDER  3
 9      8      7

 6      5      4

 3      2      1

       C=A+B
 10     10     10

 10     10     10

 10     10     10

  H
 360
```

MATRIX PROGRAM EXAMPLE 2

```
10 DIM A(20,20),B(20,20),C(20,20)
20 READ N
30 MAT A= CØN(N,N)
40 MAT B=CØN(N,N)
45 MAT C = ZER(N,N)
50 FØR I= 1 TØ N
60 FØR J = 1 TØ N
70 LET A(I,J) = 1/(I+J-1)
80 NEXT J
90 NEXT I
93 PRINT " HILBERT MATRIX ØF ØRDER "N
95 MAT PRINT A;
100 MAT B= INV(A)
105 PRINT "INVERSE ØF HILBERT MATRIX ØF ØRDER "N
110 MAT PRINT B;
115 MAT C=A*B
117 PRINT "HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER"N
118 MAT PRINT C;
120 GØ TØ 20
190 DATA 2,3
1999 END

RUN

HILTST

 HILBERT MATRIX ØF ØRDER  2
 1      .5

 .5        .333333

INVERSE ØF HILBERT MATRIX ØF ØRDER  2
 4.       -6.

-6.        12.

HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER 2
 1.        0

-3.72529 E-9     1.

 HILBERT MATRIX ØF ØRDER  3
 1      .5        .333333

 .5        .333333  .25

 .333333  .25       .2

INVERSE ØF HILBERT MATRIX ØF ØRDER  3
 9.       -36.       30.

-36.       192.     -180.

 30.      -180.      180.

HILBERT MATRIX TIMES ITS ØWN INVERSE ØRDER 3
 1.       -1.78814 E-7     0

-2.23517 E-8     1.       -5.96046 E-8

-1.49012 E-8    -1.78814 E-7     1


ØUT ØF DATA IN  20
```

3. ALPHANUMERIC DATA AND STRING MANIPULATION

Alphanumeric data, names, and other identifying information can now be handled in the BASIC language using string variables. Now you can input, store, compare and output alphanumeric and certain special characters in the GE-265 character set.

A STRING is any sequence of alphanumeric and certain special characters in the GE-265 character set not used for control purposes in the GE-265 system.

STRING size is limited to 15 valid characters.

A STRING VARIABLE is denoted by a letter followed by a "$". For example: A$, B$, X$

3.1 THE DIM STATEMENT

Strings can be set up as one-dimensional arrays only. Requests for two-dimensional arrays are not allowed and when encountered will initiate the error comment DIMENSION TOO LARGE.

Examples:

```
10 DIM A(5),C$(20),A$(12),D(10,5)

20 DIM R$(35)

30 DIM M$(15),B$(15)
```

In statement 10, only C$ and A$ are string variables. R$, as dimensioned in statement 20, will set aside space in core for 35, 15 character arrays. Any or all of these strings may be less than 15 characters.

3.2 THE LET STATEMENT

Strings and string variables may appear in only two forms of the LET statement. The first is used to replace a string variable with the contents of another string variable:

Example:

```
56 LET G$ = H$
```

and the second is used to assign a string to a string variable:

Example:

```
60 LET J$ = "THIS STRING"
```

Arithmetic operations may not be performed on string variables. Requests for addition, subtraction, multiplication or division involving string variables produce the error message ILLEGAL STRING OPERATION AT XXX.

3.3 THE IF-THEN STATEMENT

Only one string variable is allowed on each side of the IF-THEN relation. All of the six standard relations (=, <>, <, >, <=, >=) are valid. When strings of different lengths are compared, the shorter string and the corresponding part of the longer string will be used. If they compare, the shorter string is taken to be the lesser of the two.

Examples:

```
100 IF N$="SMITH" THEN 105

200 IF A$<>B$ THEN 205

300 IF "JUNE"<=M$ THEN 305

400 IF D$>="FRIDAY" THEN 600
```

The collating sequence used in comparing alphanumeric information is listed in Table I. Characters are compared in their BCD representations.

TABLE I. COMPARISON ORDER

BCD	CHARACTER	BCD	CHARACTER	BCD	CHARACTER
00	0	24	D	53	$
01	1	25	E	54	*
				*(55)	End of Message
02	2	26	F	56	>
03	3	27	G	57	↑
04	4	30	H	60	(space)
05	5	31	I	61	/
		*(32)	Bell		
06	6	33	. (period)	62	S
07	7	34	" (quote)	63	T
10	8	35	?	64	U
11	9	36	<	65	V
		*(37)	Carriage Ret.		
12	' (apostrophe)	40	- (minus)	66	W
13	:	41	J	67	X
14	(	42	K	70	Y
15	;	43	L	71	Z
				*(72)	Line Feed
16	=	44	M	73	, (comma)
17	\	45	N	74	)
20	+	46	O	75	[
21	A	47	P	76	]
22	B	50	Q	*(77)	Fill
23	C	51	R		
		*(52)	Tab		

*BCD codes of nonprinting characters are enclosed in parentheses.

3.4 READ, INPUT AND DATA STATEMENTS

READ and INPUT statements can contain string variables intermixed with ordinary BASIC variables. In the respective DATA and executable statements, every item corresponding to a string variable in an input list must be a valid string. If the string contains any characters that have special meaning in the BASIC language – such as commas, semicolons, leading or trailing spaces, etc. – it must be enclosed by quotation marks. Unquoted strings must also begin with alphabetic characters.

Examples:

```
1000 READ A5,A$,Z$,Q$

1010 INPUT L$(17),M$,NS(I)

2000 DATA 17, SHOPPING, DAYS, "LEFT"
```

3.5 THE PRINT STATEMENT

The PRINT statement also can contain string variables intermixed with ordinary BASIC variables. When a string variable is encountered which has not been assigned, the PRINT statement will produce a string of 15 spaces. A semicolon after a string variable in a PRINT statement causes the string to be printed and the variable following that string to be directly connected to it.

Examples:

```
35 PRINT A,16,B$,C$;N

40 PRINT 100+I,"DATA",L$;M$;N$

50 PRINT S$
```

4. COMPUTED GO TO STATEMENT

The computed GO TO statement has been added to BASIC, providing a multi-branched switch. The form of the statement is:

ON expression GO TO ln_1, ln_2, ...

where: expression is a valid BASIC expression.
ln_1, ln_2, ... is a sequence of line numbers to which the statement will transfer depending on the expression value.

For example:

```
ON X+Y GO TO 575, 490, 650
```

The above statement will transfer control to 575, 490, or 650 respectively, depending upon whether the value of the expression X+Y yields 1, 2, or 3 respectively.

The expression value will be truncated to its integer value if it is not already an integer. For example, if X+Y = 2.5 it will be truncated to 2 and the program will branch to the second line in the list.

5. MULTIPLE VARIABLE REPLACEMENT

The LET statement now permits multiple variable replacement.

For example:

```
LET X=Y=Z=21*N/2
```

The statement places the value of the expression "21*N/2" in variables X, Y, and Z. Any valid expression may be used.

6. TAB PRINT FUNCTION

The PRINT statement now permits tabbing of the teletypewriter. Whenever the print function TAB is used in the PRINT statement, it will cause the print head to move over to the position indicated by the argument of TAB. For example:

```
PRINT X; TAB (N); Y; TAB (2*N); Z
```

The statement will cause the print head to move over to the Nth position after printing the value of X and to the (2*N)th position after printing the value of Y.

The use of the comma remains unchanged in the PRINT statement. The semi-colon causes a slightly tighter packing than previously. Thus, when a comma follows a variable in a PRINT statement, a fixed field width is reserved before the next entry in the statement is recognized. The semi-colon causes this field width to be minimized. Thus, when the teletype is being tabbed, the semi-colon should be used.

If the argument of TAB is less than the current teletypewriter position, it is ignored.

All arguments of TAB are modulo 75. Teletypewriter print positions are assumed to run 0 through 74.

7. SGN FUNCTION

The function SGN (argument) yields +1, -1, or 0 depending upon the value of the argument. The following table describes the options:

Function	Argument Value	Yield
SGN	Zero	0
SGN	positive, non-zero	+1
SGN	negative, non-zero	-1

Examples:

SGN (0) yields 0

SGN (-1.82) yields -1

SGN (989) yields +1

SGN (-.001) yields -1

SGN (-0) yields 0

8. RND FUNCTION

The function RND is a psuedo random number generator. It requires a single argument which is meaningful in the following manner:

1. If the argument is positive, the argument is used to initiate the random number sequence.
2. If the argument is zero, RND will supply a random number. The first use of RND(0) in a program will always yield the same random number. This option is identical to the RND in BASIC prior to these extensions.
3. If the argument is negative, a random number will be used to initiate a random number sequence.

Options 1 or 3 would probably be used to initiate a sequence of random numbers after which option 2 would be used repeatedly.

9. TIME-OF-DAY CLOCK FUNCTION

The function CLK(X), where X is a dummy argument, yields the time of day in military hours.

Examples:

```
100 PRINT CLK(X)
296 IF CLK(X) > 15.00 THEN 1000
130 IF A - CLK(X) < .5 THEN 1000
```

10. PROGRAM ELAPSED TIME FUNCTION

The function TIM(X), where X is a dummy argument, yields the program elapsed time in seconds.

Examples:

```
100 PRINT TIM(X)
299 IF TIM(X) >10 THEN 1000
```

11. CHAIN FUNCTION

The CHAIN statement allows you to stop the execution of the current program and begin compilation and execution of another program without direct intervention. The CHAIN statement is equivalent to giving the commands STOP, OLD, a program name, and RUN.

The statement form is:

CHAIN saved program name

or

CHAIN saved program name, line number

Examples:

```
100 CHAIN NEXT
      or
100 CHAIN NEXT, 100
      or
100 CHAIN PLOTER***
```

```
      or
100 CHAIN TUT01$***
      or
100 CHAIN PAYROL, 555
```

Only one program name may appear in a statement. The name must conform to the rules used in naming BASIC programs. Notice that BASIC library programs may be chained as shown in the examples above.

When a number appears after the saved program name, as in the second and fifth lines above, the number indicates the line number of the named program at which execution is to begin. When no number appears, execution begins with the first executable statement of the named program.

Once a CHAIN statement is executed, the current program is stopped and the named program brought in. Because there is no logic path to any statements following the CHAIN statement, all needed current program statements must be executed before the CHAIN statement.

12. CALL STATEMENT

The CALL statement is used to call an external program for use as a subroutine within the main program just as the GO SUB statement calls a subroutine inside the main program.

The statement form is:

CALL saved program name

Examples:

```
10 CALL HISDWN
20 CALL EQCLS*
```

You may call previously saved programs of your own (line 10), common programs in your catalog library (line 20), or system library programs, either regular or run-only.

```
40 CALL A B
50 CALL AB
```

The standard program naming rules apply. Statements 40 and 50 both call a program named AB, since the BASIC system ignores all leading, trailing, and imbedded blanks. No arguments are permitted after the program name in the CALL statement. Subroutines may call other routines but no program may call the main program or itself.

All variables and defined functions are common to the main program and the called subroutines. They need not be defined separately in each program.

Example:

```
MAIN

60 DEF FNP(Z) = SQR (X ↑ 2 + Y ↑ 2 + Z ↑ 2)
70 CALL DEFPRT
80 PRINT A

DEFPRT

90  LET Y = 4
100 LET X = 7
110 LET A = FNP(3)
120 RETURN
```

Statement 70 calls DEFPRT, which stores 4 in Y and 7 in X and calculates a value for A by use of a function defined in the main program (line 60). The function uses the value 3 for Z as defined in statement 60. DEFPRT then returns to the statement immediately following the call statement, and the calculated value for A (8.60233) is printed.

The return from a subroutine to the calling program is by a RETURN statement. Multiple returns are permissible. The return is always to the statement immediately following the statement in which the program was called.

An END or a STOP statement terminates all execution, whether the statement is executed in a subroutine or in the main program.

The line numbers in the different programs are completely independent. GO TO and IF-THEN statements reference line numbers in their own program only.

Data is compiled from the main program first, and then from each of the called programs in the order in which the CALL statements are encountered. Consider the following example:

```
MAIN

120 READ A, B
130 CALL SUBR
140 READ C, D, E
150 DATA 1, 2
160 DATA 3, 4, 5

SUBR

170 READ X, Y, Z
180 DATA 6, 7, 8
190 RETURN
```

Statement 120 reads the numbers 1 and 2 into A and B. Statement 130 transfers control to program SUBR which reads 3, 4, and 5 (not 6, 7, and 8) into variables X, Y, and Z. After the return to statement 140, 6, 7, and 8 will be read into C, D, and E.

The CALL statement allows more effective use of program space available in BASIC programs. A program referred to by a CALL statement will be compiled only once no matter how many times it is called in each of the routines. Object code generated by the called program counts toward object code limitation but the characters in the called program do not count toward the BASIC character limitation. As many as 10 different programs may be called.

When an error occurs in a subroutine, the name of the subroutine is printed on the same line as the error message. The compiler error messages and their possible causes are listed below.

ERROR MESSAGE	INTERPRETATION
ILLEGAL PROGRAM NAME IN XXXXX	More than six characters or no characters in a program name.
	A program tries to call itself or the main program.
OVER 10 SUBROUTINES	An attempt was made to call more than 10 different programs.
PROGRAM NOT SAVED (Program Name)	Self explanatory.
PROGRAM WON'T FIT (Program Name)	The program called is too large to fit in the remaining available space.

13. DATA FILE CAPABILITY

Under program control, data files can be read from or written onto the disc to remain as permanent records of information for further processing by either the same or other programs. Additionally, the end-of-data condition may be tested, files may be restored to their starting point, and files may be backspaced to the previous data item during reading.

Data files are comparable to program files which the user creates from a remote terminal in the usual time-sharing way. Thus all of the editing commands such as LIST, EDIT DELETE, etc. may be used in accessing and modifying data files.

13.1 INITIAL FILE PREPARATION

Reading. If a file (or files) with initial values is to be read from the disc, you must prepare it before program execution and save it in your catalog.

Example:

```
NEW

NEW FILE NAME -- RFILE

READY.

10 1, 1.5, 2, 2.5, 3
20 3.5, 4, 4.5, 5, 5.5

SAVE
READY.
```

When preparing a disc data file, you may, at your option, use a blank in place of a comma as a data separator. Note that the word DATA is not needed in these files. The first number on each line is the line number. RFILE in the above example can be prepared as:

```
10 1 1.5 2 2.5 3
20 3.5 4 4.5 5 5.5
```

This option allows files prepared in FORTRAN to be processed in BASIC and conversely.

Writing. If a file is to be written during program execution, an area on the disc large enough to contain the entire file to be generated must be reserved in your library before program execution. This is done by renaming and saving special files available in the system library (see 13.10). These files are essentially empty. Their purpose is to preset the size of the files to be generated.

Example:

```
OLD
OLD FILE NAME -- CH0768***
READY.

RENAME
NEW FILE NAME -- WFILE
READY.

SAVE
READY.
```

13.2 FILE REFERENCE

The general form of the file reference statement is:

FILES $name_1$; $name_2$; $name_3$; . . . . ; $name_n$

where $name_1$, $name_2$, etc. are the names of files to be read or written by the program. The file reference statements must precede all executable statements in the program. The number of files that may be used in a program depends on the size of the program. You may always reference as many as 8 files in any program. If the program is a small one, you may reference as many as 50 files.

All files referenced in the FILES statements will be established as read mode files. Before a user can write into a file he must "scratch" it by issuing a SCRATCH command. This protects the user against the accidental destruction of valuable data files.

The named files must be saved in your catalog before running the program. Files which are not saved cause the error message FILE NOT SAVED to be printed.

File naming must conform to the established conventions for naming programs except:

- File names must not contain semicolons. (Semicolons will be interpreted as file name separators.)
- Leading and imbedded blanks are ignored.
- The file name is left-justified.
- File names should not contain slashes, "/", or commas ",".

Examples:

```
10 FILES A; B; C
      or
10 FILES A
20 FILES B; C
```

The following file names are identical:

```
10 FILES XYZ
10 FILES XY Z
10 FILES X Y Z
```

13.3 FILE DESIGNATOR

The file designator is used in all file commands. It points to a file named in the file referencing statement. It may be an integer, an expression, a variable, or a subscripted variable.

Example:

```
10 FILES F1; F2
      :
      :
70 READ # 1, A (I), B(I)
      :
      :
95 READ # A*B, A(I), B(I)
```

Statement 70 refers to file F1. Statement 95 refers to a file depending on the value of the expression A*B, which must be integral. If the value of A*B is 1, file F1 is selected; if the value of A*B is 2, file F2 is selected.

If the value of the file designator is less than one, non-integral, or greater than the number of files referenced, the error message ILLEGAL FILE DESIGNATOR will be printed.

13.4 FILE MODES

All user files to be processed by BASIC must be considered as either read mode or write mode. Initially, the FILES statement results in all files being set to read mode. This protects the user from accidentally destroying valuable data files. The user can subsequently change the mode of any file by issuing a SCRATCH or RESTORE statement. The SCRATCH statement establishes the designated file as WRITE mode. The RESTORE statement establishes the designated file as READ mode.

13.5 FILE READING

The general form of the read file statement is:

READ # file designator, input list

where the file designator is as described in 13.3.

The input list consists of the variables, separated by commas, into which the data is to be read. The list may contain non-string variables and string variables, any of which may be subscripted.

Example:

A file containing 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5 is to be read into A(I) and B(I)

```
10 FILES RFILE
20 FOR I = 1 TO 5
30 READ # 1, A(I), B(I)
40 NEXT I
```

For each execution of the READ # command one value is read into an A(I) and B(I) so that, in the example, at the termination of the loop:

A(1) = 1	B(1) = 1.5
A(2) = 2	B(2) = 2.5
A(3) = 3	B(3) = 3.5
A(4) = 4	B(4) = 4.5
A(5) = 5	B(5) = 5.5

The file will remain positioned following the last read data item (5.5 in the example) until further file commands designating RFILE are executed.

13.6 FILE WRITING

The general form of the write file statement is:

WRITE # file designator, output list

where the file designator is as described in 13.3.

The output list consists of the variables, separated by semicolons, from which the data file is generated. The list may contain non-string variables, string variables, strings, and expressions. Subscripting is permissible in the output list.

WRITE # always generates a file beginning with line number 1000, incrementing it by 10 for each new line. This line number sequencing can be modified, if you wish, by EDIT RESEQUENCE.

Each WRITE # statement generates one line of output unless the teletypewriter line limit is exceeded or the last list item is followed by a semicolon. When the teletypewriter line limit is exceeded writing continues on the next line with the next data item. When the output list is followed by a semicolon, subsequent writing occurs on the same line in a closely packed format.

Example:

```
100 FILES WFILE
105 SCRATCH # 1
110 FOR I = 1 TO 25
120 WRITE # 1, I; I*I
130 NEXT I
```

When listed WFILE would contain:

```
1000      1      1
1010      2      4
1020      3      9
  .       .      .
  .       .      .
  .       .      .
1240      25     625
```

The following is an example of how strings and string variables are used in conjunction with files.

```
10 FILES STRING
15 SCRATCH # 1
20 LET A$ = "STRING1"
30 WRITE #1, A$; "STRING2"
40 WRITE #1, "STRING2"; A$
```

Then listing the file STRING would yield:

```
1000 STRING1 STRING2
1010 STRING2 STRING1
```

13.7 END-OF-FILE AND END-OF-SPACE

The end-of-file (EOF) is a special mark written by BASIC itself which indicates the logical end of data in the file.

The end-of-space (EOS) is the physical end of the disc area reserved for a file. When a file is saturated with data, EOF = EOS, otherwise EOF < EOS.

Whenever a word is generated with a WRITE # statement, an EOF mark is placed immediately following the last word written. Subsequent transmissions to the file effectively erase or "bump" the EOF mark so that the EOF mark always follows the last word written. When a file is generated from the teletype, an EOF mark is placed immediately after the last data item as soon as the file is saved.

13.8 END-OF-FILE TEST

The general form of the statement which tests for an EOF mark or the EOS is:

IF END # file designator THEN line number

where the file designator is as described in 13.3.

This command will test whether an EOF/EOS condition was detected by the last command reading the designated file. When writing a file, this command will test whether an EOS condition was detected.

If the last READ # command encountered an EOF mark or EOS condition, the program will go to the line number specified in the IF END # command. Otherwise the next sequential command is executed.

If the program continues reading or writing a file after the EOF/EOS condition has been detected, an error message (END OF FILE or END OF SPACE) will be printed and the program will continue executing. The error message will be repeated each time an attempt is made to exceed the EOF/EOS limit.

Example:

```
10 FILES F1; F2
15 SCRATCH # 2
20 DIM X(100), Y(100), Z(100)
30 FOR I = 1 to 100
40 IF END # 1 THEN 80
50 READ # 1, X(I), Y(I), Z(I)
60 WRITE # 2, SQR (X(I) ↑ 2 + Y(I) ↑ 2 + Z(I) ↑ 2)
70 NEXT I
80 STOP
```

If file F1 contained 300 data items, no EOF would be encountered, but if it contained only 150 data items, for example, the IF END # statement (line 40) would cause a transfer to line 80.

13.9 FILE RESTORING

The general form of the restore file statement is:

RESTORE # file designator

where file designator is as described in 13.3.

This command causes the position of the designated file to be moved so that the next transmission is from the beginning of the file.

If a file is already restored, the restore file command merely sets the file mode to read.

The programming language automatically restores the referenced files before the program begins executing.

Example:

```
10 FILES UTIL
15 SCRATCH # 1
20 FOR I = 1 TO 50
30 WRITE # 1, I, SQR (I)
40 NEXT I
50 RESTORE # 1
          .
          .
          .
70 READ # 1, X, X1
```

In this example the RESTORE # statement (line 50) is required to restore the written file before reading it. Reading then begins at the first data item in the file.

A file being written cannot be read before it is restored. A file being read cannot be written before it is scratched. This means that if any modifications via the program are to be made to an existing file, it must be copied (written) to another file up to the modification point. The modification can then be written into the second file.

13.10 FILE SCRATCHING

The form of the scratch file statement is:

SCRATCH # file designator

where file designator is as described in 13.3. This command causes the file designated to be "scratched" or made ready for writing. If the file is already reset or restored this command merely sets the file mode to write. Following the SCRATCH statement, transmission to the file commences at the beginning of the file.

Example:

```
10 FILES UTIL
20 SCRATCH # 1
30 FØR I = 1 TO 50
40 WRITE # 1, RND (0)
50 NEXT I
60 RESTØRE # 1
70 READ # 1, X,Y,Z
```

In this example the SCRATCH # statement (line 20) must be executed before the write into the file. The RESTORE # statement must be executed before reading from the written file.

13.11 FILE BACKSPACING

The general form of the backspace file statement is:

BACKSPACE # file designator

where file designator is as described in 13.3. This command is permitted only on files in the read mode.

The backspace command causes the position of the file being read to be moved backward one data item. If the file is already at its start point, the backspace statement is ignored.

Some applications require a data file to be processed forward and then backward. The following example illustrates how this can be done.

Example:

```
10 FILES F1
20 IF END # 1 THEN 110
30 READ # 1, A
35 REM COUNT NUMBER OF POINTS IN FILE
40 LET N = N+1
.          .
.          .
.          .

100 GO TO 20
110 FOR I = 1 TO N
120 BACKSPACE # 1
130 READ # 1, A
140 BACKSPACE # 1
.          .
.          .
.          .
200 NEXT I
```

13.12 DUMMY CATALOG FILES

Since it is necessary to save a data file of a size large enough to contain the data to be written, the

system library contains the following 6 files which can be renamed as desired and saved in your catalog.

LIBRARY NAME	CHARACTERS*	LIBRARY NAME	CHARACTERS*
CH0192***	192	CH1536***	1536
CH0384***	384	CH3072***	3072
CH0768***	768	CH6144***	6144

*The number of data points that can be written into a file is a function of the number of characters in each data point. The line number and spaces must also be considered in the count.

13.13 FILE ERROR MESSAGES

ERROR MESSAGE	INTERPRETATION
FILE(S) NOT SAVED:	The files indicated have been referenced but are not saved in your library.
FILES NOT FIRST	The file reference statement is preceded by an executable statement.
FILES NOT DEFINED	The file reference statement is missing.
TOO MANY FILES	More than 50 files have been referenced or the number of files referenced caused the program size limit to be exceeded. (At least 8 files will always be allowed.)
READING FILE	An attempt has been made to write into a read mode file. Indicates a logic error or SCRATCH command encountered before read mode activity.
WRITING FILE	An attempt has been made to read or backspace a write mode file. Indicates a logic error or no RESTORE encountered before read mode activity.
DATA FILE (LINE XXXXX) FORMAT ERROR	At line XXXXX of the data file being read, data is not in the required format.
END OF FILE	An attempt has been made to read data from a file after all data has been read. The file pointer is located at the end of file. No data is transmitted.
END OF FILE SPACE	After all physical space in a file has been exhausted, an attempt has been made to write into the file. No data is transmitted.
ILLEGAL FILE DESIGNATOR	File designator is less than unity, non-integral, or greater than number of referenced files.

All of the above messages, except TOO MANY FILES, are followed by the line number of the statement causing the error.

14. INITIALIZATION

In Extended BASIC all variables are initialized to zero.

Computer Centers and offices of the Information Service Department are located in principal cities throughout the United States.

Check your local telephone directory for the address and telephone number of the office nearest you. Or write . . .

General Electric Company
Information Service Department
7735 Old Georgetown Road
Bethesda, Maryland 20014

GENERAL ELECTRIC

INFORMATION SERVICE DEPARTMENT

802207 (25M) 4-68