EXTENDED BASIC

User's Manual

093–000065–06

Ordering No. 093-000065

Printed in the United States of America
Rev. 06, February 1975

NOTICE

Original Release-November, 1971
First Revision -May, 1972
Second Revision -September, 1972
Third Revision -March, 1973
Fourth Revision -September, 1973
Fifth Revision -January, 1975
Sixth Revision -February, 1975

This revision of the Extended BASIC User's Manual, 093-000065-06, constitutes a major revision and supersedes all previous revisions and addenda.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

CHAPTER 1

INTRODUCTION

GENERAL INFORMATION

Extended BASIC provides programmers with an interactive programming language that operates under Data General's Real Time Disk Operating System (RDOS) or Data General's Real Time Operating System (RTOS). Extended BASIC may be configured to include:

- Disks
- Floating Point Hardware
- Swapping
- Fixed Point Multiply/Divide
- Mapping
- Single or Double Precision

Data General's Extended BASIC is an implementation of the BASIC language as developed at Dartmouth College. Extended BASIC includes such features as:

- String Manipulation
- Format Control
- Assembly Language Subroutines
- Matrix Operations
- Fixed and Variable Length File Manipulation
- Program and Keyboard Modes

The following Data General documents may be referred to for further information:

093-000075 RDOS Real Time Disk Operating System User's Manual

093-000083 Introduction to the Real Time Disk Operating System

093-000119 Extended BASIC System Manager's Guide

093-000087 BATCH User's Guide

093-000056 Real Time Operating System Reference Manual

DOCUMENTATION CONVENTIONS

Underlining

Where clarification is required in the examples used in this manual, underlined copy denotes entries input by the user. Copy not underlined indicates entries output by BASIC.

Symbols

The symbols listed below are used throughout this manual to simplify descriptions.

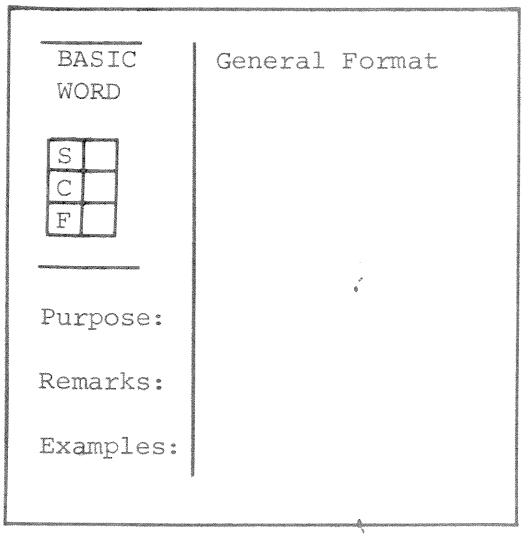
Symbol	Character Represented	Explanation
)	Carriage Return	Pressing the RETURN key generates an automatic line feed in addition to the carriage return.
ESC	ESCape or ALTmode key	When using BASIC, pressing the ESCape key echoes an asterisk (*) on the user's terminal.
Δ	Space	Sometimes used in this manual to emphasize a space character.
[ ]	Brackets	Enclose optional arguments to a statement or command.
{ }	Braces	Indicate a choice of items enclosed.
...	Elipsis	Indicate that the preceding item may be repeated.

Terminal Devices

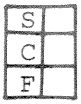
The use of the word "terminal" throughout this manual implies a teletypewriter, CRT display terminal, or an equivalent interactive device.

Statement and Command Descriptions

The statements and commands described in Chapters 3 through 9 of this manual are presented as follows:



General Format: The BASIC word is shown in its generalized format with capital letters used to indicate literal entries, lowercase italics used for variable entries, and brackets used to indicate optional arguments. Parentheses are to be inserted as indicated.



: The appropriate box or boxes are checked to indicate whether the BASIC word is used as a statement (S), a command (C), or a function (F). Some BASIC words may be used both as statements and commands.

Purpose: A brief statement which describes the operation performed by the statement, command or function.

Remarks: Pertinent comments related to the use of the statement, command, or function. Any rules or cautions are included under this heading.

Statement and Command Descriptions

(Continued)

Examples: Typical uses are provided to help describe the BASIC word and its format.

Abbreviations

The following abbreviations are used in the general formats provided in the descriptions of BASIC statements and commands. The abbreviations are italicized in the formats and represent commonly used terms which are defined in the appropriate chapters of the manual.

Abbreviation	Term
var	numeric variable
expr	numeric expression
rel-expr	relational expression
string lit	string literal
val	numeric values
line no.	line number
col	column
control var	control variable
svar	string variable
mvar	matrix variable
filename	a disk file name or a device

Terminology

The BASIC language includes words, sometimes referred to as keywords or instructions which, when written in an appropriate format, can be used as statements and functions in a program or as console commands to the BASIC system.

Some BASIC words can be used alone to perform an operation. Others require one or more *arguments* in order to be properly executed.

```
INPUT A,B        ← A and B are arguments to
                   the INPUT instruction.
```

A BASIC program is made up of BASIC statements. Each statement includes a properly formatted BASIC word preceded by a line number in the range 1 to 9999. The line number given to a BASIC statement determines the order in which it is executed. Generally, program execution begins

Terminology (Continued)

with the lowest numbered line and is followed sequentially by the next higher numbered line unless otherwise directed by statements such as GOTO or GOSUB.

Each program statement is written on a separate line. The programmer terminates each line with a carriage return (↵).

```
* 5  PRINT "SAMPLE PROGRAM"
*10  LET A=5
*15  LET B=2
*20  PRINT A*B
*25  END
```

BASIC console commands do not include line numbers and are executed by the system immediately after the user terminates the command with a carriage return.

RUN↵ BASIC executes the user's current program starting from the lowest numbered line.

USING
EXTENDED BASIC

Logging On

The user can log onto the system as soon as the BASIC prompt message DGC READY is output to the terminal. The log on procedure is begun when the user presses the ESCape key. During this procedure the system will request the user's identification. The user responds by typing his or her four character identification code, assigned by the System Manager, followed by a carriage return (↵). The identification is not echoed at the terminal to protect the confidentiality of the user's identification.

If the identification entered is valid, the system will output the date, time and terminal number assigned, followed by an asterisk (*) prompt.

Logging On

(Continued)

The asterisk (*) prompt is used by BASIC to signal that the user may enter a command or a program statement.

The format of the log on procedure is as follows:

```
DGC READY

ESC                              ← user presses ESC.

ACCOUNT ID:  XXXX)               ← 4 character ID,
                                   not echoed.

ΔΔΔΔΔMM/DD/YYΔΔΔΔΔHH:MMΔSIGN-ON,ΔZZ
        |        |        |
      date     time   terminal no.

*                                ← asterisk prompt.
```

Creating A New Program

Having successfully logged onto the system, the user may enter a new program, make corrections to, or run an old program. It is generally good practice to type the NEW command before proceding with entering a new program. This command (NEW) clears the user's work area in memory and thereby prevents the interspersion of lines from previous programs into the user's new program. This command, as well as other system and interactive commands used for such purposes as retrieving and storing programs, is described in Chapter 8.

When typing a new program, the user must be certain to begin each line with a line number of not more than four digits and end each line with a carriage return. If a typing error is detected before the carriage return is pressed, the user can correct it by pressing the RUBOUT key once for each character to be erased until the incorrect character is reached and then continue by typing the correct characters. Each time the RUBOUT key is pressed, a backarrow (←) is echoed at the terminal. For example:

```
* 1Ø  PRINT "CONR←←RRECTION BY RUBOUTS"
```

Creating A New Program

(Continued)

In line number 10, the user rubbed out two characters (R and N) and then completed the line. A LIST command would output the corrected line.

```
* LIST 10)
0010 PRINT "CORRECTION BY RUBOUTS"
*
```

In addition, the user may delete the entire current line by typing SHIFT-L which is echoed as a backslash (\) and a carriage return.

```
*10 PRINT "CONR\          (SHIFT-L to delete line)
10 PRINT "CORRECTION BY RUBOUT" (line typed over)
```

The SHIFT-L character may also be used to delete the current console command line.

```
*RU\          (SHIFT-L to delete line)
```

Running A Program

When the user has completed typing a program, it can be executed by giving the command:

```
RUN)
```

The program will be run starting from the lowest numbered statement, assuming there are no runtime system errors (see Appendix A) and will output results requested via PRINT statements.

Programs which were previously written and SAVEd may be run by typing:

```
*LOAD filename)        (filename is the name of a
*RUN)                   user's program)
```

Correcting The Program

After running a program, the user may find it necessary to change the program because of error messages or incorrect results. Corrections can be made to the program by any of the following procedures:

a. A new statement may be substituted for a statement containing errors by retyping the entire line (including line number).

b. A statement may be eliminated from the program by typing its line number followed by a carriage return.

```
* 125)          (line 125 is deleted)
```

c. Additional statements may be inserted into the program between existing statements simply by typing the new statements with intermediate line numbers. If the number of statements to be inserted exceeds the number of line numbers available between the statements, it may be necessary to use the RENUMBER command (described in Chapter 8) to change the increment between line numbers. It is generally good practice when writing a program, to allow an increment of 10 between line numbers for program correction and expansion.

Interrupting Program Execution

To stop the execution of a running program, the listing of a program, or any other task which is being performed by BASIC, the user may press the ESCape key to interrupt the process. BASIC will then output an asterisk prompt to signal the user that a new command may be entered.

Logging Off

Having completed working with BASIC at the terminal, the user logs off by typing the command BYE. The BASIC system will then output a summary of usage information and put the terminal in an idle state.

Logging Off

(Continued)

```
*BYE)
ΔΔΔΔΔMM/DD/YY  HH:MM  SIGN-OFF, ZZ
     MM/DD/YY  HH:MM  CPU USED, QQ
     MM/DD/YY  HH:MM  I/O USED, RR

DGC READY
```

where:

MM/DD/YY	is today's date.
HH:MM	is the current time of day.
ZZ	is the terminal port number.
QQ	is the number of CPU seconds consumed during the terminal session calculated to the nearest tenth of a second.
RR	is the number of input and output statements executed (OPEN, CLOSE, READ, WRITE, etc.)

Example Of An Extended BASIC Program

The following example is shown in a manner which includes the logging in, communicating with the system operator, running the program, and logging off by the user.

```
DGC READY
ESC                          (Press ESC key)
ACCOUNT-ID:  KAST)           (KAST not echoed)
     10/23/74      10:32 SIGN-ON, 2
*MSG OPER PLS MOUNT TAPE #1255 (NO RING))
*FROM OPER:  DONE-TAPE ON MT12
*MSG OPER THANX)
*LOAD "PRODUCTION")

*LIST)
0010 DIM A$(10)
0020 INPUT "TAPE MOUNTED ON",A$
0030 A$=A$,":0"
0040 OPEN FILE (0,3),A$
0050 READ FILE (0),A,B,C$
0060 IF EOF(0)=1 GOTO 200
0070 PRINT A,B,C$
```

(Continued on next page)

Example Of An Extended BASIC Program

(Continued)

```
0100 GOTO 50
0200 CLOSE FILE(0)
0210 PRINT "END OF JOB"
0220 STOP

*RUN)
TAPE MOUNTED ON MT12)
END OF JOB

STOP AT 0220
*MSG OPER PLS RELEASE MT12)
*FROM OPER:  TAPE REMOVED FROM MT12
*BYE)
     10/23/74      10:40  SIGN-OFF, 2
     10/23/74      10:40  CPU-USED, .3
     10/23/74      10:40  I/O-USED, 0

DGC READY
```

CHAPTER 2

EXTENDED BASIC ARITHMETIC

NUMBERS

An extended BASIC number may be in the range of $5.4 * 10^{-79} < N < 7.2 * 10^{75}$. Numbers may be expressed as integers, floating point or in exponential form (E-type notation).

BASIC provides either all single precision or all double precision calculations. The format of converted numeric data (for example, as converted by a PRINT statement) is dependent upon the BASIC system generated.

Single Precision Calculations

On conversion, any floating point or integer number that consists of six digits, or less, is formatted without using exponential form. A floating point or integer number that requires more than six digits is printed in the following E-type notation.

(sign)n.nnnnnE(sign)XX

Where n.nnnnn is an unsigned number carried to five decimal places with trailing zeros suppressed, E means "times 10 to the power of", XX represents an unsigned exponential value.

Number	Single Precision Output Format
2,000,000	2E+06
108.999	108.999
.0000256789	2.56789E-05
24E10	2.4E+11

Double Precision Calculations

On conversion, any floating point or integer number that consists of eight digits, or less, is formatted without using exponential form. A floating point or integer number that requires more than eight digits is printed in the following E-type notation.

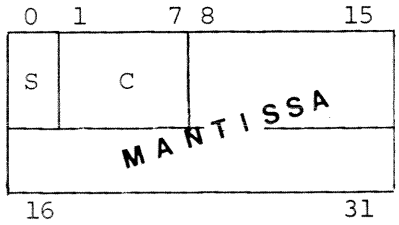
(sign)n.nnnnnnnE(sign)XX

where n.nnnnnnn is an unsigned number carried to 7 decimal places with trailing zeros suppressed, E means "times 10 to the power of", and XX represents an unsigned exponential value.

Number	Double Precision Output Format
.666666666	.6666667
108.999868	108.99987
111111111.99	1.1111111E+08

Internal Number Representation

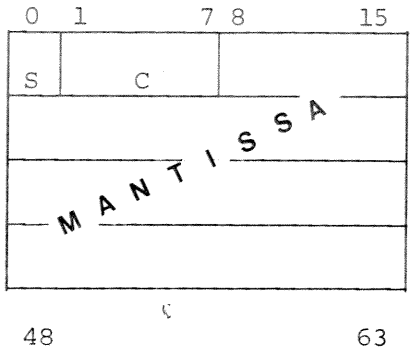
Internally, BASIC stores numbers in a format compatible with other Data General Corporation software such as FORTRAN IV and the relocatable assemblers. Single precision floating point numbers are stored in two consecutive 16-bit words of the form:



where: S is the sign of the mantissa.
0 = positive, 1 = negative.
The mantissa is a normalized six digit hexadecimal fraction.
C is the characteristic and is an integer expressed in excess 64 code.

Internal Number Representation (Continued)

Double precision floating point numbers add 2 words of precision to the mantissa, which can be represented as :



VARIABLES

The names of numeric variables (shown in program statements as *var*) are expressed as either a single letter or a single letter followed by a digit. For example :

Acceptable Variable Names	Unacceptable Variable Names
A	6A
A3	AZ
Z	
Z6	

In addition to numeric variables, string variables (*svar*) are also permitted in BASIC and are discussed in Chapter 5.

ARRAYS

An array represents an ordered set of values. Each member of the set is called an array element. An array can have either one or two dimensions. An array name may be a single letter, or a single letter followed by a digit.

Array Elements

Each of the elements of an array is identified by the name of the array followed by a parenthesized subscript.

```
B3(1) , B3(2) , ..., B3(8) , B3(9)
```

For a two-dimensional array, the first number gives the number of the row and the second gives the number of the column for each element. The elements of array C(2,3) would be:

```
C(1,1)   C(1,2)   C(1,3)

C(2,1)   C(2,2)   C(2,3)
```

A reference to element zero (Ø) will be interpreted as a reference to element 1. A negative reference is an error.

If a variable is referenced both with and without subscripts, then two distinct variables will be defined by BASIC. For example:

```
*10 DIM A1 [11]
*20 LET A1 = 17
*30 LET A1(1) = 27
```

In all subscripting contexts, brackets ([]) may be used in place of parentheses [()].

Declaring an Array

Most arrays are declared in a DIM statement, which gives the name of the array and its dimensions.

The lower bound of a dimension is always 1; the upper bound is given in the DIM statement. Dimensional information is enclosed in either parentheses or square brackets immediately following the name of the array in the DIM statement.

```
*5 DIM A(15), B1[2,3]
```

There is no limitation on the number of elements in a given array dimension other than restrictions due to available memory.

Declaring an Array (Continued)

If an array is not declared in a DIM statement then a default value of 10 is assigned to each dimension of the array.

```
*10 C[3,4] = 11
```

If C has not appeared in a DIM statement which was executed prior to the execution of line 1Ø, then when line 1Ø is executed C will be given dimensions [10,10].

ARITHMETIC OPERATIONS

A numeric expression (shown in program statement formats as *expr*) can be composed of numbers, numeric variables, array variables and functions, linked together by arithmetic operators. The operators used in writing numeric expressions are:

Operator	Meaning	Example
+	Unary plus	A+(+B)
-	Unary minus	A+(-B)
↑	Exponentiation	A↑B (A to the B power)
*	Multiplication	A*B
/	Division	A/B
+	Addition	A+B
-	Subtraction	A-B

See Chapter 5 for string arithmetic operators.

Priority of Arithmetic Operations

BASIC evaluates numeric expression (*expr*) in the following order proceeding from left to right:

1. Any *expr* within parentheses are evaluated before any unparenthesized *expr*. When parenthesized *exprs* are nested, the innermost *expr* is always evaluated first.

2. Unary plus and minus

3. Exponentiation

4. Multiplication and division (equal priority)

5. Addition and subtraction (equal priority)

Priority of Arithmetic Operations (Continued)

6. When two operators are of equal precedence (* and /), evaluation proceeds from left to right.

For example:

```
Z - A + B * C ↑ D
```

Step 1. A is subtracted from Z
Step 2. C ↑ D is evaluated
Step 3. B is multiplied by the result of Step 2.
Step 4. result of step 3 is added to the result of Step 1.

Use of Parentheses

Since parenthesized *exprs* are evaluated first, the programmer can use parentheses to change the order of evaluation for an *expr*. Using the same variables as in the previous example;

```
Z - ( ( A + B) * C) ↑ D
```

Step 1. A + B is evaluated.
Step 2. The value from step 1 is multipled by C.
Step 3. The value from step 2 is raised to the D power.
Step 4. The value from step 3 is subtracted from Z.

Parentheses may also be used to clarify the order of evaluation and legibility of an *expr*. For example, the following *exprs* are equivalent:

```
A * B ↑ 3/4 + B/C + D ↑ 3

((A*B↑3)/4) + ((B/C) + D ↑ 3)
```

Relational Operators and Expressions

Relational operators are used to compare two *exprs* in a relational-expression (*rel-expr*). A relational expression is of the form:

expr1 relational operator expr2

Relational Operators and Expressions (Continued)

The relational operators used in BASIC are:

Symbol	Meaning	Example
=	Equal	A = B
<	Less than	A < B
<=	Less than or equal	A <= B
>	Greater than	A > B
>=	Greater than or equal	A >= B
<>	Not equal	A <> B

Strings may also be used in the place of the *expr* in relational expressions. Their usage is described in Chapter 5.

CHAPTER 3

COMMONLY USED BASIC STATEMENTS

COMMENTING A PROGRAM

REM

REM [*message*]

message: text comment.

S	√
C	
F	

Purpose: To insert explanatory comments within a program.

Remarks: REM statements are ignored when the program is executing. However, the REM statement is stored with the program and is output exactly as entered when LISTed.

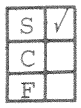
If control is transferred to a REM statement from a GOTO or GOSUB statement, then execution continues with the next executable statement following the REM statement. If no executable statement follows the REM statements then the program will act as though an END statement were encountered and control will return to interactive mode.

Examples:

```
*10 REM -- REMARKS THROUGHOUT A PROGRAM CAN
*20 REM -- HELP EXPLAIN THE PURPOSE OF STATEMENTS.
*30 REM -- LINES 10, 20, 30 ARE NOT EXECUTED.
```

STOPPING EXECUTION OF
A PROGRAM

END



END

Purpose:

To terminate execution of the program and to return control to interactive mode.

Remarks:

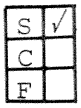
Data General's implementation of Extended BASIC does not require the inclusion of an END statement to declare the physical end of a program. If control passes through the last executable statement of the program and if that statement does not change the flow of control (that is, the statement is not a GOTO, etc.) then the program will transfer control to interactive mode. The END statement is included in this implementation for compatibility with BASIC programs written for other systems. Multiple END statements may appear in the same program, and when encountered will terminate execution of the program followed by a prompt (*) printed at the user's terminal.

Examples:

```
*20 PRINT "PROGRAM DONE"
*30 GOTO 60
    .
    .
    .
*50
*60 END
* RUN
PROGRAM DONE

END AT 0060
*
```

STOP

STOP



Purpose:

To terminate execution of the program and to return control to interactive mode.

Remarks:

STOP statements may be placed anywhere in the program to terminate execution. When STOP is encountered in the program the system will print the following message on the user's terminal:

```
STOP AT XXXX
*
```

where XXXX is the line number of the STOP statement.

After resumption of interactive mode, the program may be restarted in its initial state (see RUN) or continued in its current state (see CON or RUN *line number*).

Examples:

```
*LIST
0010 REM--TERMINATE PROGRAM BY STOP
0020 INPUT A
0030 IF A<0 THEN GOTO 0050
0040 GOTO 0020
0050 STOP

*RUN
 ? 1
 ? 3
 ? -5

STOP AT 0050
*
```

ASSIGNMENT STATEMENT

LET

[LET] *var* = *expr*

var: numeric variable name.
expr: an arithmetic expression.

S	✓
C	✓
F	

Purpose:

To evaluate *expr* and assign the resultant value to *var*.

Remarks:

Use of the mnemonic LET is optional.

The variable *var* may be subscripted.

String expressions may be assigned to string variables (see Chapter 5).

Examples:

```
* 10 LET A=A+1
```

Variable A is assigned a value one greater than it was before.

```
* 20 A(2,1) = B↑2+10
```

The element in row 2/column 1 of array A is assigned the value of expression B↑2+10.

INPUT STATEMENTS

Input statements are used to define and read data that is to be used during program execution.

INPUT

S	✓
C	✓
F	

INPUT ["*string lit*",] *var* [,*var*]...[;]

var: a list of variables separated by commas.

string literal: a message or prompt. (See Chapter 5 for detailed string information.)

Purpose:

To assign the values supplied by input from the user's terminal to a list of variables.

Remarks:

1. The INPUT statement may be used to enter numeric data, string data (see Chapter 5) or both to a program.

2. When an INPUT statement is executed, a question mark (?) is output as an initial prompt unless the INPUT statement contains the *"string literal"* option. Then the *"string literal"* is output as an initial prompt.

3. The user responds by typing a list of data, where each datum is separated from the next by a comma or a carriage return. The list is terminated with a carriage return.

4. If the data list is terminated with a carriage return before a value has been supplied for each of the elements of the variable list, then a question mark (?) will be output as a prompt, indicating there are further data list elements which must be supplied.

Remarks:
(Continued)

5. The data input in response to the prompt must be of the same mode (numeric or string) as the variable in the INPUT statement list for which the data is being supplied. Variables in the INPUT statement list may be subscripted or unsubscripted.

6. If data input from the terminal does not match the mode of a variable in the INPUT statement list, then a \? is output to the terminal for the data in error.

7. If the variable list is terminated with a semi-colon, then the cursor is left following the last input data item. Otherwise, a carriage return-line feed is output.

Examples:

1.
```
*LIST
0005 INPUT A,B,C,D,E
0010 PRINT A+B,C+D,D+E

*RUN
 ? 1,2,3,4,5
 3              7              9

END AT 0010
*
```

2.
```
*LIST
0010 INPUT "A,B,C,D,E= ",A,B,C,D,E
0020 PRINT A+B,C+D,D+E

*RUN
A,B,C,D,E= 1,2 ? 3,4,5
 3              7              9

END AT 0020
*
```

Examples:
(Continued)

3.
```
*LIST
0010 INPUT A,B,C;
0020 PRINT "  NO RETURN"

*RUN
 ? A\ ? 1,2,3  NO RETURN

END AT 0020
*
```

DATA

S	✓
C	
F	

DATA $\begin{Bmatrix} val \\ "string\ lit" \end{Bmatrix} \left[\begin{Bmatrix} ,val \\ ,"string\ lit" \end{Bmatrix} \right] \ldots$

val and *string lit:* a list of numeric values and string literals.

Purpose: To provide values for variables appearing in READ statements.

Remarks: The DATA statement is a non-executable statement. The values appearing in a DATA statement or statements form a single list.

The first element of this list is the first item in the lowest numbered DATA statement. The last item in this list is the last item in the highest numbered DATA statement.

Both numbers and string literals (see Chapter 5) may appear in a DATA statement and each value in the DATA statement list must be separated from the next value by a comma.

Examples:

```
100 DATA 1, 17, "AB,CD", -1.3E-13
```

(See the READ and MAT READ statements for usage and additional examples.)

READ

S	√
C	√
F	

READ $\left\{\begin{matrix} var \\ svar \end{matrix}\right\} \left[\left\{\begin{matrix} ,var \\ ,svar \end{matrix}\right\}\right] \ldots$

var and *svar*: a list of numeric and string variables separated by commas.

Purpose:

To read values from the data list (DATA statements) and assign them to the variables listed in the READ statement.

Remarks:

1. READ statements are always used in conjunction with DATA statements.

2. The variables listed in the READ statement may be subscripted or non-subscripted and may be numeric or string (see Chapter 5).

3. The order in which variables appear in the READ statement is the order in which values for the variables are retrieved from the data list.

4. A data element pointer is moved to the next available value in the data list as values are retrieved for variables in READ statements. If the number of variables in the READ statement exceeds the number of values in the data list, an END OF DATA error message is printed.

5. The mode (numeric or string) of the READ statement variable must match the mode of the corresponding DATA element value or a READ/DATA TYPES error message is printed.

6. The RESTORE statement can be used to reset the data element pointer to the first item of the lowest numbered DATA statement or to the first item of a particular DATA statement.

Examples:

```
*LIST
0010 READ A,B,C
0020 READ D[1],D[2],D[3]
0030 PRINT C↑2,D[2]↑2
0040 READ E
0050 PRINT E
0060 READ F$
0070 PRINT F$
0080 DATA 1,2,3,4,5,6,7,"ABC"
0090 END

*RUN
 9              25
 7
ABC

END AT 0090
*
```

In this example the variables are assigned values as follows:

Variable	Value
A	1
B	2
C	3
D(1)	4
D(2)	5
D(3)	6
E	7
F$	ABC

RESTORE

S	✓
C	✓
F	

RESTORE [*line no.*]

line no.: a DATA statement line number.

Purpose:

To reset the position of the data element pointer.

Remarks:

If the RESTORE statement is used without a line number argument, then the data element pointer is reset to the beginning of the data list.

If the RESTORE statement is used with a DATA statement line number argument, then the data element pointer is positioned to the first value in the DATA statement line.

Examples:

```
* 5  READ A,B,C
* 10 READ D,E,F
* 15 RESTORE 50
* 20 READ G,H,I
* 25 RESTORE
* 30 READ J,K,L
* 40 DATA 2,4,6
* 50 DATA 8,10,12
```

In the above example the variables are assigned values as follows:

Variable	Values
A	2
B	4
C	6
D	8
E	10
F	12
G	8
H	10
I	12
J	2
K	4
L	6

OUTPUT STATEMENTS

Output statements are used to print the results of your program at the terminal.

PRINT

S	✓
C	✓
F	

$$\left\{\begin{matrix};\\ \text{PRINT}\end{matrix}\right\}\left[\left\{\begin{matrix}expr\\ "string\ lit"\end{matrix}\right\}\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\left\{\begin{matrix}expr\\ "string\ lit"\end{matrix}\right\}\right]\ldots\right]\left[\left\{\begin{matrix},\\ ;\end{matrix}\right\}\right]$$

Semicolon (;): a substitute for keyword PRINT.
expr: a numeric expression.
string lit: a message or prompt. (See Chapter 5 for detailed string information.)

Purpose:

To perform one of the following print operations on the user's terminal:

1. Print the result of a computation.
2. Print verbatum the characters in a string literal.
3. Print a combination of uses 1 and 2.
4. Print a blank line (skip a line).

Remarks:

Printing Numbers

Numbers (integer, decimal, or E-type) are printed in the following form:

sign number space

The sign is either minus (-) or blank for plus and the number is always followed by a blank space. (See Chapter 2 for further details on numeric formats).

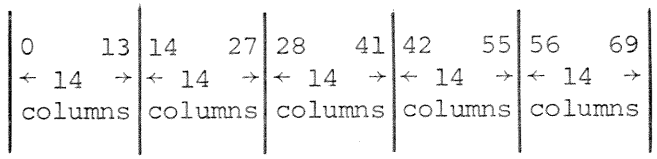
Zone Spacing of Output

The print line on a terminal is divided in print zones. The width of a print zone is determined by the TAB command described in Chapter 8. The

Remarks:
(Continued)

Zone Spacing of Output (Continued)

default value for TAB is 14 and is used in the following examples. The first column number on a line is column 0.



A comma (,) between items in the PRINT statement list causes the next item to be printed in the leftmost position of the next printing zone. If there are no more printing zones on the current line, printing continues in the first printing zone on the next line. If a list element requires more than one print zone, the next item in the list is printed in the next free print zone (see example 1).

Before each list item is printed its length is compared with the space remaining on the line. If insufficient space is left on the current line the item is moved to the next line. If the length of the item is greater than the width of the page (see PAGE command in Chapter 8) then an error message is issued.

Compact Spacing of Output

A semicolon (;) between items in the PRINT statement list causes the next item to be printed at the next character position. Note that a space is always printed after a number and that a space is reserved for the plus (+) sign even though it is not printed. (See example 2.)

PRINT

Remarks:
(Continued)

Spacing To The Next Line

When the last item in a print list has been printed, a carriage return and line feed is output unless the last item in the list is followed by a comma (,) or semicolon (;). In this case the carriage return and line feed are not output and the next item is printed on the same line according to the comma or semicolon punctuation. (See example 3.)

Printing Blank Lines

A PRINT statement with no list of print items or punctuation will cause a carriage return and line feed to be output. (See example 4.)

Additional printing versatility can be accomplished by use of the TAB(X) function, the TAB= command, The PAGE= command the PRINT USING statements described in Chapters 8 and 9.

Examples:

1.
```
*LIST
0010 LET X=25
0020 PRINT "THE SQUARE ROOT OF X IS:",SQR(X)

*RUN
THE SQUARE ROOT OF X IS:      5

END AT 0020
*
```

```
↑              ↑              ↑
0             14             28
                        (column positions)
```

Examples:
(Continued)

2.
```
*LIST
0010 LET X=5
0020 PRINT X;(X*2)↑6;X*2;(X*2)↑4;
0030 PRINT X-25;(X*2)↑8;X-100

*RUN
 5   1E+06   10   10000 -20   1E+08 -95

END AT 0030
*
```

```
↑    ↑       ↑    ↑       ↑    ↑      ↑
0    4       11   15      21   26     32
                              (column positions)
```

Lines 20 and 30 use the semicolon (;) form for keyword PRINT and then use the semicolon as the spacing character.

3.
```
*NEW
*5 PAGE=70
*10 LET X=5
*20 PRINT X,(X*2)↑6,
*30 PRINT X↑4
*40 PRINT "FIN"
*RUN
 5                1E+06           625
FIN

END AT 0040
*
```

```
↑                ↑                ↑
0               14               28
                        (column positions)
```

Notice that the trailing comma in line 20 causes the value of X↑4 in line 30 to be printed in zone 3 rather than zone 1.

Examples:
(Continued)

4.
```
*LIST
0010 LET X=5
0020 PRINT X;(X*2)↑6,X*2
0030 PRINT X-25;(X*2)↑8
0040 PRINT X-100
0050 PRINT
0060 PRINT "DONE"

*RUN
 5   1E+06        10
-20   1E+08
-95

DONE

END AT 0060
*
```

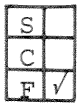
```
↑     ↑              ↑
0     4             14    (column positions)
```

At line 20, the comma and semicolon spacing characters are both used. Line 50 outputs a blank line before "DONE".

TAB(X)

TAB(*expr*)

expr: an expression which is evaluated to an integer.



Purpose:

The TAB(X) function, which may only be used in PRINT statements, tabulates the print position to the column number evaluated from *expr*.

Remarks:

1. Columns are numbered 0 through 71 for conventional terminals. More than one TAB(X) function may appear in a PRINT statement and the column number indicated by the function is always relative to column 0. The position at which the next item in the print list is printed will depend on the value of *expr* and on the punctuation (; or,) following the TAB(X) function.

2. If *expr* evaluates to a column number greater than or equal to the current column and less than the width of the page, then the new current column position becomes the value of the expression. If the TAB function is followed by a semi-colon (;) then no change is made in the value of the current column following evaluation of the TAB function. If a comma (,) follows the TAB function and if the current column position is at the beginning of a zone then no further changing occurs. Otherwise the current column position is set to the start of the next zone. After the determination of the current column the next PRINT list item is output. (See PRINT statement remarks.)

3. If *expr* evaluates to a column number lower than the present column number, the TAB(X) function is ignored, and positioning proceeds as in 2.

Remarks:
(Continued)

4. If *expr* evaluates to a column number greater than the carriage length, the expression is reduced modulo the carriage length and positioning proceeds as in 2.

5. If *expr* evaluates to 0, then TAB(0) causes a carriage return and line feed and positioning proceeds as in 2.

Example:

```
*LIST
0005 LET A=-6
0010 LET B=5
0015 PRINT  TAB(B);A; TAB(2*B);2*A
0020 END

*RUN
     -6    -12

END AT 0020
*
```

```
↑     ↑     ↑
0     5     10        (Column positions)
```

Notice the use of the semicolon (;) in line 15 after "A" to prevent spacing to the next print zone and passing position 2*B (Column 10).

DIMENSIONING
ARRAYS

DIM

S	✓
C	✓
F	

DIM { *array (m)* / *array (row,col)* } [{ , *array (m)* / , *array (row,col)* }] ...

array: a BASIC numeric identifier.
m: the number of elements in a one dimensional array.
row: the number of rows in the array.
col: the number of columns in the array.

Purpose:

To explicitly define the size of one or more numeric variable arrays. Dimensioning of string arrays is discussed in Chapter 5.

Remarks:

Array Elements

The concept of arrays is described in Chapter 2. The DIM statement is used to declare the size of an array to be a number of elements other than the default number (10) for each dimension.

```
* 10 DIM A(13),B(7,7),C(20,5)
```

The initial value of all elements in an array is zero until assigned a value by the user's program.

Any variable or expression that is used for a subscript must evaluate to a value in the range:

1≤value≤upper bound declared in DIM statement

```
*  5 X=2
* 10 PRINT A(1,X↑2)
```

If the variable or expression subscript does not evaluate to an integer, BASIC will convert it using the INT function (See Chapter 4).

Remarks:
(Continued)

Array Elements (Continued)

If a subscript evaluates to an integer larger than the upper bound of the dimension for the array or smaller than 0, the subscript error message is printed.

Redimensioning Arrays

It is possible to redimension a previously defined array during execution of a program by declaring the array in another DIM statement. The total number of elements of the newly dimensioned array must not exceed the previous total number of elements.

```
* 100 DIM A(3,2)
    .
    .
    .
* 200 DIM A(2,3)
    .
    .
    .
* 300 DIM A(2,2)
```

The values assigned to elements in array A(3,3) are reassigned to elements in array A(2,3) and then to elements in array A(2,2).

1 2 3	1 2 3	1 2
4 5 6	4 5 6	3 4
7 8 9		
A(1,1) = 1	A(1,1) = 1	A(1,1) = 1
A(1,2) = 2	A(1,2) = 2	A(1,2) = 2
A(1,3) = 3	A(1,3) = 3	A(2,1) = 3
A(2,1) = 4	A(2,1) = 4	A(2,2) = 4
A(2,2) = 5	A(2,2) = 5	
A(2,3) = 6	A(2,3) = 6	
A(3,1) = 7		
A(3,2) = 8		
A(3,3) = 9		

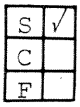
PROGRAM LOOPS

Programs which require the repetitive operation of a block of statements until a termination condition is met can be simplified by use of a FOR - NEXT program loop.

A program loop begins with a FOR statement which provides the specifications for repetition, a block of statements which is executed during each repetition of the program loop, and a NEXT statement which denotes the end of the loop.

```
FOR statement
(block of statements)
NEXT statement
```

FOR and NEXT



FOR *control var* = *expr1* TO *expr2* [STEP *expr3*]
(Block of statements)
NEXT *control var*

control var:	a non-subscripted numeric variable.
expr1:	a numeric expression which defines the first or initial value of the *control variable*.
expr2:	a numeric expression which defines the terminating value of the *control variable*.
expr3:	a numeric expression which defines the increment added to the *control variable* each time the loop is executed.
(Block of statements):	any statements which may also contain FOR - NEXT loops.

Purpose:

To establish the initial, terminal and incremental values for a *control variable* which is used to determine the number of times a block of statements contained in a FOR - NEXT loop are to be executed. The loop is repeated until the value of the *control variable* meets the termination condition.

Remarks:

Rules

1. *control var*iable must not be subscripted.

2. Every FOR or NEXT statement must have a matching NEXT or FOR statement or an error message is printed.

3. Expressions *expr1*, *expr2* and *expr3* may have positive or negative values and *expr3* must not be zero.

4. If STEP *expr3* is omitted from the FOR - NEXT statement, then *expr3* is assumed to be +1.

5. The termination condition for a FOR - NEXT loop is dependent upon the values of *expr1* and *expr3*. The loop terminates if: *expr3* is positive and the next value of *control var* is greater than *expr2*; *expr3* is negative and the next value of *control var* is less than *expr2*.

   If the value of *expr1* (the initial value) meets the termination condition, then the loop is not performed even once.

6. If the body of a FOR - NEXT loop is entered at any point other than the FOR statement, then, upon encountering the NEXT statement corresponding to the skipped FOR statement an error message will be issued.

7. When the termination condition is met, the loop will be exited and the value of the *control var* will be final value of *control var*.

Remarks:
(Continued)

8. A loop may be exited using a GOTO or GOSUB statement before the *control variable* has met the termination condition.

Program Loop Operation

1. The expressions *expr1*, *expr2* and *expr3* are evaluated. If *expr3* is not specified it is assumed to be +1.

2. The *control var* is set equal to *expr1*.

3. If *expr3* is positive and *control var*>*expr2* then the termination condition is satisfied and control is passed to the statement following the corresponding NEXT statement.

   If *expr3* is negative and *control var*<*expr2* then the termination condition is satisfied and control is passed to the statement following the corresponding NEXT statement.

   Otherwise, the following steps are performed.

4. The statements in the FOR - NEXT block are executed.

5. When the corresponding NEXT statement is executed, *control var* is set equal to *control var + expr3*.

6. Repeat step 3.

Nesting Loops

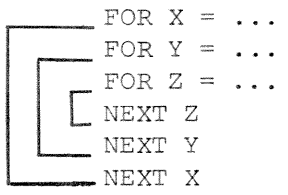
FOR - NEXT loops may be nested to a depth specified by the system manager. The FOR statement and its terminating NEXT statement must be completely contained within the loop in which it is nested. For example:

FOR and NEXT (Continued)

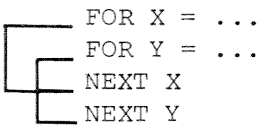
Remarks: (Continued)

Nesting Loops (Continued)

Legal Nesting



Illegal Nesting



Examples:

1.
```
*LIST
0010 FOR I=1 TO 9
0020 NEXT I
0030 PRINT I

*RUN
 9

END AT 0030
*
```
← I equal last value assigned during execution of loop.

2.
```
*LIST
0040 FOR J=1 TO 9 STEP 3
0050 NEXT J
0060 PRINT J

*RUN
 7

END AT 0060
*
```
← Final value of J before terminating value was exceeded.

Examples:
(Continued)

3.
```
*LIST
0010 LET P=3.14159
0020 FOR I=0*P/180 TO 360*P/180 STEP 45*P/180
0030   PRINT  TAB(30+10*SIN(I));"X"
0040   FOR J=1 TO 3
0050     PRINT  TAB(30);"*"
0060   NEXT J
0070 NEXT I

*RUN
                              X
                              *
                              *
                              *
                                    X
                              *
                              *
                              *
                                       X
                              *
                              *
                              *
                                    X
                              *
                              *
                              *
                              X
                              *
                              *
                              *
                         X
                              *
                              *
                              *
                       X
                              *
                              *
                              *
                         X
                              *
                              *
                              *
                            X
                              *
                              *
                              *

END AT 0070
*
```

Examples: (Continued)

4.
```
*LIST
0010 FOR I=1 TO 3 STEP -1
0020   PRINT "SHOULD NOT ENTER HERE"
0030 NEXT I
0040 PRINT I

*RUN
 1

END AT 0040
*
```

SUBROUTINES

A subroutine is a group of program statements which is entered via the GOSUB statement and exited via the RETURN statement. Rather than repeat the statements at each point they are required, the statements are written into the program only once and are accessed by a GOSUB statement. The RETURN statement allows control to return to the statement following the last GOSUB statement. In this manner, the program continues at the appropriate place after the subroutine has been executed.

GOSUB and RETURN

GOSUB *line no.*
RETURN

S	✓
C	
F	

line no.: a line number.

Purpose:

GOSUB directs program control to the first statement of a subroutine. RETURN exits the subroutine and returns program control to the next statement following the GOSUB statement that caused the subroutine to be entered.

Remarks:

1. A subroutine may only be entered by using a GOSUB statement. Otherwise, the RETURN-NO GOSUB error message is printed when the RETURN statement is executed.

2. A subroutine may have more than one RETURN statement should program logic require the subroutine to terminate at one of a number of different places.

3. Although a subroutine may appear anywhere in a program, it is good practice to place the subroutine distinctly separate from the main program. In order to prevent inadvertant entry to the subroutine by other than a GOSUB statement, the subroutine

Remarks:
(Continued)

should be preceded by a STOP statement or GOTO statement which directs control to a line number following the subroutine.

4. Subroutines may be nested to a depth specified by the system manager. Nesting occurs when a subroutine is called during the execution of a subroutine. On execution of a RETURN statement, control is passed to the statement immediately following the most recently executed GOSUB statement.

Examples:

1.
```
*LIST
0010 LET A=6
0020 GOSUB 0100
0030 LET A=10
0040 GOSUB 0100
0050 STOP
0100 FOR I=1 TO A STEP 2
0110   PRINT I;
0120 NEXT I
0130 PRINT
0140 RETURN

*RUN
 1  3  5
 1  3  5  7  9

STOP AT 0050
*
```

Examples:
(Continued)

2.

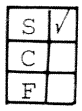
```
*LIST
0010 GOSUB 0040
0020 PRINT " EXAMPLE"
0030 STOP
0040 PRINT "NEST";
0050 GOSUB 0080
0060 PRINT "INE";
0070 RETURN
0080 PRINT "ED";
0090 GOSUB 0120
0100 PRINT "ROUT";
0110 RETURN
0120 PRINT " SUB";
0130 RETURN

*RUN
NESTED SUBROUTINE EXAMPLE

STOP AT 0030
*
```

BRANCH STATEMENTS

The following statements permit branching from one portion of a program to another. The GOTO statement is unconditional and provides branching to the line number specified in the statement. The ON-GOTO/GOSUB and IF-THEN statements are conditional and branching occurs on the basis of evaluated conditions.

GOTO

GOTO *line no.*



line no.: a program statement line number.

Purpose:

To unconditionally transfer control to a statement that is not in normal sequential order.

Remarks:

1. If control is transferred to an executable statement, that statement and those following will be executed.

2. If control is transferred to a non-executable statement (e.g., DATA) program execution will continue at the first executable statement which follows the non-executable statement.

Examples:

(Continued on next page)

Examples:
(Continued)

```
*LIST
0010 READ X
0020 PRINT X
0030 GOTO 0010
0040 DATA 1,2,3,4,5
0050 DATA 20,21,23
0060 END

*RUN
 1
 2
 3
 4
 5
 20
 21
 23

ERROR 15 AT 0010 - END OF DATA
*
```

IF -- THEN

S	√
C	√
F	

IF { *rel-expr* / *expr* } THEN *statement*

rel-expr: a relational expression as defined in Chapter 2.
expr: a numeric expression.
statement: any BASIC statement except FOR, NEXT, DEF, END, DATA and REM.

Purpose:

To execute a statement on the basis of whether an expression or a relational expression is true or false.

Remarks:

1. If, after evaluation, the relational expression, *rel-expr*, is true, then the program statement following the THEN is executed. If the relation is false, program execution continues at the next sequential statement after the IF--THEN statement.

2. A numeric expression (*expr*) may be used in place of a relational expression. The numeric expression is considered false if it has a value of 0 and is true if it has a non-zero value.

Note: Since the internal representation of non-integer numbers may not be exact (for example .2 can not be exactly represented), it is advisable to test for a range of values when testing for a non-integer. For example, if the result of a computation, A, was to be 1.0 a reliable test for 1 is

```
IF ABS (A-1.0)<1.0E-6 THEN...
```

If this test succeeds, then A is equal to 1 to within 1 part in 10↑6. This is approximately the accuracy of single precision floating point calculations.

IF -- THEN

Examples:
(Continued)

1.
```
* 10  IF A=B THEN GOTO 100
* 20  IF A=B GOTO 100
* 30  IF A-B <= 5 THEN C=0
* 40  IF A*B < 50 THEN GOSUB 300
* 50  IF A↑B > 100 GOSUB 400
```

Lines 10 and 20 are equivalent variations of the IF -- THEN statement.

2.
```
*LIST
0005 REM---START
0010 LET N=10
0020 INPUT "X=",X
0030 IF X THEN GOTO 0050
0040 GOTO 0100
0050 IF X>=N THEN GOTO 0080
0060 PRINT X,"X IS LESS THAN 10"
0070 GOTO 0020
0080 PRINT X,"X GREATER OR EQUAL TO 10"
0090 GOTO 0020
0100 PRINT X,"X=0"
0110 END

*RUN
X=5
 5              X IS LESS THAN 10
X=7
 7              X IS LESS THAN 10
X=12
 12             X GREATER OR EQUAL TO 10
X=10
 10             X GREATER OR EQUAL TO 10
X=0
 0              X=0

END AT 0110
*
```

Examples:
(Continued)

3.
```
*LIST
0010 LET X=5
0020 LET A$="12ABC34"
0030 IF X=5 THEN IF A$[3,X]="ABC" THEN PRINT "SUPER"
0040 END

*RUN
SUPER

END AT 0040
*
```

This example compares strings in the relational expression. See Chapter 5 for detailed string information.

ON-GOTO
ON-GOSUB

ON *expr* {GOTO / GOSUB} *line no.* [,*line no.*]...

S	✓
C	
F	

expr: a numeric expression which is evaluated to an integer.

line no.: a list of line numbers in the current program whose positions in the argument list are numbered from 1 through n.

Purpose:

To transfer control to one of several lines in a program depending on the computed value of an expression at the time the statement is executed.

Remarks:

1. The expression *expr* is evaluated and if it is not an integer, the fractional portion is ignored.

2. The program transfers control to the line number whose position in the argument list corresponds to the computed value of *expr*.

3. If *expr* evaluates to an integer that is greater than the number of lines given in the argument list or that is less than or equal to zero, the ON statement is ignored and control passes to the next statement.

4. The ON-GOSUB statement must contain an argument list whose lines are the first line of subroutine within the current program.

Example:

```
*10  ON  M-5  GOTO  500,75,1000
```

If M-5 evaluates to 1, 2 or 3 then control will transfer to statement 500, 75 or 1000, respectively. If M-5 evaluates to any other value, control will transfer to the next sequential BASIC statement in the program.

CHAPTER 4

EXTENDED BASIC FUNCTIONS

INTRODUCTION TO EXTENDED BASIC FUNCTIONS

Extended BASIC provides functions to perform calculations which eliminate the need to write programs to perform these calculations. The functions generally have a three character mnemonic name and are followed by a parenthesized expression (*expr*) which is the function argument. Generally, a function may be used as an expression, or may be included as part of an expression.

The following extended BASIC functions are described in this chapter.

Function	Value Produced
RDN(X)	Random number between 0 and 1
SGN(X)	The algebraic sign of X
INT(X)	The integer value of X
ABS(X)	Absolute value of X
SQR(X)	Square root of X ($X \geq 0$)
EXP(X)	e^X ($-178 \leq X \leq 175$)
LOG(X)	Natural logarithm of X ($X > 0$)
SIN(X)	Sine of X (X expressed in radians)
COS(X)	Cosine of X (X expressed in radians)
TAN(X)	Tangent of X (X expressed in radians)
ATN(X)	Arctangent of X (result expressed in radians)
SYS(X)	System functions
FNa(d)	User defined function

In addition, there are a number of functions which are described in other chapters of this manual which relate to strings, matrices and files.

INTRODUCTION
TO EXTENDED
BASIC FUNCTIONS
(Continued)

Function		Refer to Chapter
TAB(X)	Printing Function	3
LEN(X$) POS(X$,Y$,Z) STR$(X) VAL(S$)	String Functions	5
EOF(X)	File Function	7

RND(X)

RND(*expr*)

expr: a numeric expression (required, but not used).

S•	
C	
F	√

Purpose:

To produce a pseudo-random number N, such that $0 \leq N < 1$.

Remarks:

The RND function requires an argument (*expr*), although the argument does not affect the resulting random number nor does the RND function affect the argument.

The RND function, each time it is called, provides a pseudo-random number in the range 0 to 1. The sequence in which these numbers is provided is fixed. The length of the sequence is 2↑16 for single and double precision arithmetic.

Since the sequence of pseudo numbers is fixed, and the start point in the sequence is reset to the same point each time a NEW or RUN is issued, the sequence of numbers provided by RND is reproducible (see RANDOMIZE for exceptions). The sequence generated on systems using double precision is different from that generated on systems using single precision.

Each occurance of the RND function yields the value of the next random number in the list.

RND(X)
(Continued)

Examples:

1.
```
*LIST
0005 TAB =10
0010 FOR I=1 TO 4
0020   PRINT RND(1),
0030 NEXT I

*RUN
 .21132     .14464     .852625    .927054
END AT 0030
*RUN
 .21132     .14464     .852625    .927054
END AT 0030
*
```

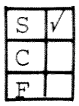
Running the above program a second time will produce the same five random numbers.

2.
```
*LIST
0005 TAB =10
0010 FOR J=1 TO 4
0020   PRINT INT(10*RND(1)),
0030 NEXT J

*RUN
 2          1          8          9
END AT 0030
*
```

This program will produce five random integers in the range 0 to 9.

RANDOMIZE

RANDOMIZE



Purpose:

To cause the random number generator to start at a different point in the sequence of random numbers generated by RND.

Remarks:

Normally the same sequence of random numbers is generated by successive use of the RND function. This feature is useful for debugging programs. When the program has been found to run successfully, the RANDOMIZE statement should be included in the program before the first occurrence of a RND function if different start points in the sequence are desired.

The RANDOMIZE statement resets the random number generator based on the time of day thereby producing different random numbers each time a program using the RND function is run.

Example:

```
*10 RANDOMIZE
*20 PRINT RND(0)
```

This program will print a different value each time it is run.

SGN(X)

SGN(*expr*)

expr: a numeric expression.

S	
C	
F	✓

Purpose:

To return a +1 if *expr* is greater than 0, a 0 if *expr* equals 0, and a -1 if *expr* is less than 0.

Example:

```
*LIST
0010 LET A=-3
0020 PRINT SGN(A)

*RUN
-1

END AT 0020
*
```

INT(X)

INT(*expr*)

expr: a numeric expression.

S	
C	
F	√

Purpose:

To return the value of the nearest integer not greater than *expr*.

Examples:

1.
```
*LIST
0010 PRINT INT(15.8)

*RUN
 15

END AT 0010
*
```

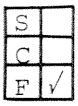
2.
```
*LIST
0010 PRINT INT(-15.8)

*RUN
-16

END AT 0010
*
```

3.
```
*LIST
0010 PRINT INT(15.8+.5)

*RUN
 16

END AT 0010
```

ABS(X)

ABS(*expr*)

expr: a numeric expression.



Purpose: To return the absolute (positive) value of *expr*.

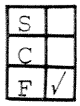
Example:

```
*LIST
0010 PRINT ABS(-30)

*RUN
 30

END AT 0010
*
```

SQR(X)

SQR(*expr*)

expr: a positive numeric expression.



Purpose: To compute the square root of *expr*.

Examples:

```
*LIST
0010 LET A=5
0020 PRINT SQR(A↑2+75)

*RUN
 10

END AT 0020
*
```

EXP(X)

EXP(*expr*)

expr: a numeric expression
(-178 $\leq$ *expr* $\leq$ 175).

S	
C	
F	✓

Purpose:

To calculate the value of e (2.71828) to the power of *expr*.

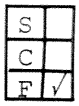
Example:

```
*LIST
0010 REM-CALCULATE VALUE OF E↑1.5
0020 PRINT EXP(1.5)

*RUN
 4.48169

END AT 0020
*
```

LOG(X)

LOG(*expr*)

expr: a numeric expression.



Purpose: To calculate the natural logarithm of *expr*.

Example:

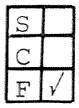
```
*LIST
0010 REM-CALCULATE THE LOG OF 959
0020 PRINT LOG(959)

*RUN
 6.86589

END AT 0020
*
```

TRIGONOMETRIC
FUNCTIONS

SIN(X)

SIN(*expr*)

expr: a numeric expression specified in radians.



Purpose:

To calculate the sine of an angle which is expressed in radians.

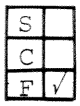
Example:

```
*LIST
0010 REM-PRINT SINE OF 30 DEGREES
0020 PRINT SIN(30*SYS(15)/180)

*RUN
 .5

END AT 0020
```

COS(X)

COS(*expr*)

expr: a numeric expression specified in radians.



Purpose:

To calculate the cosine of an angle which is expressed in radians.

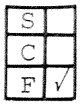
Example:

```
*LIST
0010 REM-PRINT COSINE OF 30 DEGREES
0020 LET P=SYS(15)/180
0030 PRINT COS(30*P)

*RUN
 .866025

END AT 0030
*
```

TAN(X)

TAN(*expr*)

expr: a numeric expression specified in radians.



Purpose: To calculate the tangent of an angle which is expressed in radians.

Example:

```
*LIST
0010 REM-PRINT TANGENT OF X DEGREES
0020 INPUT "X DEGREES ",X
0030 LET P=3.14159/180
0040 PRINT TAN(X*P)

*RUN
X DEGREES 45
 .999999

END AT 0040
*
```

ATN(X)

ATN(*expr*)

expr: a numeric expression.

S	
C	
F	✓

Purpose: To calculate the angle (in radians) whose tangent is *expr*. ($-\pi/2 \leq$ ATN (*expr*) $\leq \pi/2$).

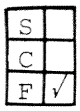
Example:

```
*LIST
0010 REM-CALCULATE ANGLE WHOSE TAN=2
0020 PRINT ATN(2)

*RUN
 1.10715

END AT 0020
*
```

SYS(X)

SYS(*expr*)

expr: a numeric value or expression.



Purpose:

To return system information based on the value of *expr* which is evaluated to an integer (0 to 16).

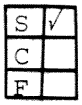
SYS(0) = the time of day (seconds past midnight)
SYS(1) = the day of the month (1 to 31)
SYS(2) = the month of the year (1 to 12) } current date
SYS(3) = the year in four digits (e.g., 1975) }
SYS(4) = the terminal port number (-1 if operator's console)
SYS(5) = CPU time used in seconds to the nearest tenth
SYS(6) = I/O usage (numbers of file I/O statements executed)
SYS(7) = the error code of the last run-time error
SYS(8) = the file number of the file most recently referenced in a file I/O statement
SYS(9) = page size
SYS(10) = tab size
SYS(11) = hours
SYS(12) = minutes } current time of day
SYS(13) = seconds }
SYS(14) = seconds remaining before expiration of timed input
SYS(15) = PI (3.14159)
SYS(16) = e (2.71828)

USER DEFINED FUNCTIONS

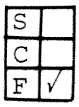
DEF

DEF FN*a*(*d*) = *expr*

a: a single letter A to Z.
d: dummy arithmetic variable that may appear in *expr*.
expr: an arithmetic expression which may contain variable *d*.



FN*a*(*d*)



Purpose:

To permit the user to define as many as 26 different functions which can be repeatedly referenced throughout a program. Each function returns a numeric value.

Remarks:

1. The dummy variable named in the DEF statement are not related to any variables in the program having the same name; the DEF statement simply defines the function and does not cause any calculation to be carried out.

2. In the function definition, the *expr* can be any legal arithmetic expression and may include other user-defined functions. Functions may be nested to a depth specified by your system manager.

3. Function definition is limited to a single line DEF statement. Complex functions which require more than one program statement should be constructed as subroutines.

Examples:

```
*LIST
0010 DEF FNE(J)=(J↑2)+2*J+1
0020 LET Y=FNE(5)
0030 PRINT Y

*RUN
 36

END AT 0030
*
```

In line 10 the FNE function is defined. In line 20 the FNE function is referenced and evaluated with numeric argument 5.

```
*LIST
0005 TAB =14
0010 LET P=3.14159
0020 DEF FNR(X)=X*P/180
0030 DEF FNS(X)=SIN(FNR(X))
0040 DEF FNC(X)=COS(FNR(X))
0050 FOR X=0 TO 45 STEP 5
0060   PRINT X,FNS(X),FNC(X)
0070 NEXT X

*RUN
 0             0              1
 5             8.71557E-02    .996195
 10            .173648        .984808
 15            .258819        .965926
 20            .34202         .939693
 25            .422618        .906308
 30            .5             .866026
 35            .573576        .819152
 40            .642787        .766045
 45            .707106        .707107

END AT 0070
*
```

This example illustrates the nesting of user defined functions.

CHAPTER 5

STRING INFORMATION

STRING CONVENTIONS

String Literals

A *string* is a sequence of characters which may include letters, digits, spaces and special characters. A *string literal* (constant) is a string enclosed within quotation marks. String literals are often used in PRINT and INPUT statements as described in Chapter 3.

```
*100 PRINT "THIS IS A STRING LITERAL"
*200 INPUT "X=",X
```

The enclosing quotation marks are not printed when the string is output to a terminal. non-printing and special characters may be included in string literals by enclosing the numeric equivalent of the character within angle brackets (< >). See Appendix E for the decimal equivalents of ASCII character codes.

```
*10 PRINT "USE DECIMAL 34 TO PRINT <34> IN STRINGS"
* RUN
USE DECIMAL 34 TO PRINT " IN STRINGS
```

String Variables

Extended BASIC permits the use of *string variables* as well as string literals. A *string variable* name consists of a letter, or a letter and a digit, followed by a dollar sign ($).

Legal String Variables	Illegal String Variables
A$	AA$
A2$	2$
D6$	3C$

String values are assigned to string variables by the use of LET, INPUT and READ statements.

Dimensioning String Variables

Unless a string variable is declared in a DIM statement, extended BASIC assumes a maximum string length of 10 characters or less. Therefore, undimensioned string variables longer than 10 characters which are used in LET, READ and INPUT statements are truncated to 10 characters. Good programming practice would suggest that all string variables be dimensioned, regardless of size.

```
*10 DIM A$ (25), B3$ (200)
```

There is no limitation on string variable size other than available memory limitations. In the DIM statement above, A$ has a maximum string length of 25 and B3$ has a maximum string length of 200.

```
*LIST
0010 DIM A2$[15]
0020 LET A2$="PRINT A2$ IS THIRTY CHARACTERS"
0030 PRINT A2$

*RUN
PRINT A2$ IS TH

END AT 0030
*
```

Substrings

Program statements which use string variables may also use portions of strings (substrings) by subscripting the string variables. Subscripted string variables are of the general form:

$$svar\ \left[\left\{\begin{matrix} x \\ y,z \end{matrix}\right\}\right]$$

svar: string variable name.
x: xth through last character of *svar*.
y,z: yth through zth characters inclusive of *svar*.

For example:

A$	References the entire string.
A$(2)	References the second character through the last character in the string inclusive.
A$(I)	References position I through the last character in the string inclusive.
A$(3,7)	References characters occupying positions 3 through 7 inclusive.
A$(I,J)	References characters occupying positions I through J inclusive, where I and J are evaluated to character positions in the string and I $\leq$ J.
A$(1,1)	References only the first character in the string.

```
*LIST
0005 DIM A1$[20]
0010 LET A1$[1,3]="SUB"
0020 LET A1$[4,10]="STRING "
0030 LET A1$[11,17]="EXAMPLE"
0040 PRINT A1$

*RUN
SUBSTRING EXAMPLE

END AT 0040
*
```

Substrings
(Continued)

String variable assignments may be changed during a program. For example:

```
*LIST
0010 LET A$="ABCDEF"
0020 PRINT A$
0030 LET B$="1"
0040 LET A$[3,3]=B$
0050 PRINT A$
0060 LET A$[4]=B$
0070 PRINT A$

*RUN
ABCDEF
AB1DEF
AB11

END AT 0070
*
```

Assigning Values To String Variables

A string variable can be assigned a string value by the use of READ and DATA statements. When string data is included in a DATA list, the string elements must always be enclosed in quotation marks.

```
*LIST
0005 DIM A1$[20],B$[10],D$[5]
0010 READ A,A1$,B$,C,D$
0015 PRINT A,C,D$
0020 DATA 5,"ABCD","EFGH",10,"IJKL"

*RUN
 5                10              IJKL

END AT 0020
*
```

As indicated by this example, string data and numeric data may be intermixed in a DATA list. However, the variables in the READ statement must match (numeric or string) the elements of the DATA list or an error message will be output.

String data may also be input to a program by the use of INPUT statements. When responding to the INPUT statement question mark (?), the use of quotation marks to enclose the string is optional. If data for more than one string variable is requested by the INPUT statement, the string data elements entered must be separated by a comma or a carriage return. Commas may be included in a string by enclosing the entire string in quotation marks. Quotation marks may be included by enclosing the value 34 in angle brackets. Caution must be exercised when NUL or CR characters are included since they are record delimiters.

```
*10 INPUT A$, B$, C, D, E$
  .
  .
  .
 RUN )
 ?ABCD, EF,GH, 2, 4, "SIX")
```

Strings in IF - THEN Statements

As mentioned in Chapter 3 (IF - THEN statement) strings may also be used in the relational expression of an IF - THEN statement. In this case, the strings are compared character by character on the basis of the ASCII character value (see Appendix E) until a difference is found. If a character in a given position in one string has a higher ASCII code than the character in that position in the other string, the first string is greater. If the characters in the same positions are identical but one string has more characters than the other, the longer string is the greater of the two.

```
*200 LET A$ = "ABCDEF"
*300 LET B$ = "25 ABCDEFG"
             .
             .
             .
*310 IF A$>B$ GOTO 500    ←True. Transfer occurs.
*320 IF A$>B$(4) GOTO 500  ←False. No transfer.
*330 IF A$(1,4)=B$(4,7) GOTO 500  ←True.
                                    Transfer occurs.
```

String Concatenation

String variables and string literals may be concatenated on the right hand side of LET statements, using a comma (,) as the concatenation operator. For example:

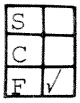
```
*100 DIM A$ (50), B$ (50)
*110 LET A$="@$2.50 EACH, THE PROFIT MARGIN IS 15.8%"
*120 LET B$=A$ (1,4), "25", A$ (7,35), "1.2%"
*130 PRINT B$
* RUN)
@$2.25 EACH, THE PROFIT MARGIN IS 11.2%.
```

STRING FUNCTIONS

A number of string functions are implemented in extended BASIC which increase string handling capabilities. The string functions are:

```
LEN (X$)
POS (X$,Y$,Z)
STR$ (X)
VAL (X$)
```

LEN (X$)



LEN (*svar*)

svar: string variable

Purpose:

To return a value equal to the number of characters currently assigned to string variable *svar*.

Remarks:

The LEN (X$) function may be used with any program statement which has an expression (*expr*) argument.

Example:

```
0005 DIM A$[80],B1$[80]
0010 INPUT A$,B1$
0020 LET B=LEN(A$)
0040 IF B>LEN(B1$) THEN GOTO 0060
0050 GOTO 0100
0060 PRINT "LENGTH OF A$=";LEN(A$)
0070 PRINT "LENGTH OF B1$=";LEN(B1$)
0080 PRINT "A$>B1$"
0090 GOTO 0110
0100 PRINT "B1$>A$"
0110 END

*RUN
 ? CHEESE ? CAKE
LENGTH OF A$= 6
LENGTH OF B1$= 4
A$>B1$

END AT 0110
```

POS (X$,Y$,Z)

POS ({*svar 1* / *"string lit 1"*}, {*svar 2* / *"string lit 2"*}, *expr*)

S	
C	
F	√

svar: string variable.
string lit: string literal.
expr: numeric expression.

Purpose:

To determine the location in a string (*svar1* or *string lit1*) of the first character of the first occurrence of a substring (*svar2* or *string lit2*) beginning at or after position *expr*.

Remarks:

The POS function returns a value equal to the first position of the substring in the string. If the substring cannot be found in the string, the POS function returns a value of zero. If the value of the starting position from *expr* is less than zero, an error message is output.

Example:

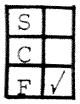
```
*LIST
0005 DIM A$[30]
0010 LET A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
0020 LET A=POS(A$,"MNOP",6)
0030 PRINT A

*RUN
 13

END AT 0030
*
```

In this example, a search is made for "MNOP" starting from the sixth character (N) in string A$. A match is found which begins at the 13th character in string A$. Therefore, the POS function returns a value of 13 which is assigned to variable A.

STR$(X)

STR$(*expr*)

expr: a numeric expression.



Purpose: To convert a numeric expression to a string which is its decimal representation.

Remarks: Converting numerics to strings permits string manipulation by other string handling functions and statements.

Example:

```
*LIST
0010 READ A
0015 IF A=0 THEN STOP
0020 LET A$=STR$(A)
0030 IF A$[4,6]="222" THEN GOTO 0050
0040 GOTO 0070
0050 PRINT A;" -THIS IS MODEL 222 FRAMISHAM"
0060 GOTO 0010
0070 PRINT A;" -THIS ISN'T OUR FRAMISHAM"
0080 DATA 111222,212222,123456,0
0090 GOTO 0010

*RUN
 111222  -THIS IS MODEL 222 FRAMISHAM
 212222  -THIS IS MODEL 222 FRAMISHAM
 123456  -THIS ISN'T OUR FRAMISHAM

STOP AT 0015
*
```

VAL (X$)

VAL ({*svar* / *"string lit"*})

S	
C	
F	✓

svar: a string variable comprised of numbers.

string lit: a string literal comprised of numbers.

Purpose:

To return the decimal representation of a string variable or string literal.

Remarks:

The string variable or string literal argument to the VAL function must consist entirely of numbers or an error message will be output. The value returned by the VAL function may be used in numeric arithmetic expressions.

Example:

```
*LIST
0010 LET A$="12345"
0020 LET B=54321
0030 LET C=VAL(A$)
0040 LET D=B+C
0050 PRINT D

*RUN
 66666

END AT 0050
*
```

STRING
ARITHMETIC

Arithmetic operations may be performed on string variables and string literals. The arithmetic operation will be executed provided the strings, or substrings which begin at the first character of the strings, have legal numeric values. Any alphanumerics which follow the numeric substring are ignored. If the substring is not a legal number, an error message is output.

Valid String	Invalid String
"123"	"FRED"
"123."	"123.E+FRED"
"-123."	"-+123"
"-123.E5"	
"-123.E-5FRED"	

Notice that decimal points, signs, and exponential format are permitted in the substring so long as they conform to the numeric representation described in Chapter 2.

The operators +, -, *, and / may be used to link strings and create an expression to be evaluated numerically. The concatenation character (,) may not be used in a string arithmetic expression.

```
*LIST
0010 LET A$="1234 GEARS"
0020 LET B$="5678 GEARS"
0030 PRINT A$+B$+"10"

*RUN
6922.

END AT 0030
*
```

Eighteen digits of precision are returned when string arithmetic calculations are made. If any precision is lost, an error message is output. For example:

```
PRINT "123E27" + "5.793E-4"
```

This statement would cause an error message since the decimal point location for the two strings causes the number of significant digits to be greater than 18.

CHAPTER 6

MATRIX MANIPULATION

DIMENSIONING MATRICES

Matrices can be dimensioned by an of three methods:

1. Using a DIM statement to declare the number of rows and columns in the matrix.
2. Including the matrix dimensions in a matrix statement.
3. Allowing a default size of 10 rows and 10 columns by not specifying dimensions in a DIM or matrix statement.

It should be noted that matrices do not have row 0 or column 0, and as in all BASIC arrays, matrix elements are stored by row in ascending locations in memory.

A number of matrix statements allow dimensioning and redimensioning so long as the new dimensions do not exceed the size of the matrix declared in a DIM or initialization statement. For example:

```
*20 DIM A(15,14)      ←210 elements in matrix A
*40 MAT A=CON(20,7)   ←140 elements
*60 MAT A=ZER(10,10)  ←100 elements
```

Statements 40 and 60, above, redimension matrix A as well as perform matrix operations described later in this chapter. The user's attention is also directed to matrix file statements in Chapter 7, File Input and Output.

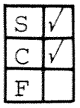
MATRIX MANIPULATION STATEMENTS

The following statements are used to copy or initialize a matrix.

Matrix Assignment

MAT *mvar*1 = *mvar*2

mvar: matrix variable name.



Purpose:

To copy the elements of matrix *mvar*2 into matrix *mvar*1.

Remarks:

This is the matrix assignment statement. Matrix *mvar*1 will assume the identical dimensions and values of matrix *mvar*2.

Example:

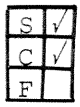
```
*LIST
0010 DIM A[2,2]
0020 LET A[1,1]=5
0030 LET A[1,2]=10
0035 MAT PRINT A
0040 MAT B=A
0050 MAT PRINT B

*RUN

 5          10
 0          0          ←Matrix A

 5          10
 0          0          ←Matrix B

END AT 0050
*
```

Line 40 will assign matrix B the same dimensions as matrix A and will also assign any element values in matrix A to the corresponding elements in matrix B. Therefore, B(1,1) = 5 and B(1,2) = 10.

Zero Matrix (ZER)

MAT *mvar* = ZER [(*row*,*col*)]

mvar: matrix variable name.
row: number of rows in matrix.
col: number of columns in matrix.



Purpose:

To set the value of each element in a matrix to zero.

Remarks:

1. The form MAT *mvar* = ZER is used for previously dimensioned matrices.
2. The form MAT *mvar* = ZER (*row*,*col*) is used if the matrix was not previously dimensioned or if the matrix is to be redimensioned.
3. The matrix elements are set to zero regardless of any previously assigned values.

Example:

```
*LIST
0005 TAB =5
0010 DIM A[3,4]
0020 LET A[1,2]=6
0030 LET A[3,4]=10
0040 MAT PRINT A
0050 MAT A=ZER[3,3]
0060 MAT PRINT A

*RUN

 0     6     0     0
 0     0     0     0
 0     0     0     10

 0     0     0
 0     0     0
 0     0     0

END AT 0060
*
```

←Matrix A after line 50.

Zero Matrix
(ZER)

Example:
(Continued)

In line 50, matrix A is redimensioned and all elements are assigned a value of zero.

Unit Matrix (CON)

S	✓
C	✓
F	

MAT *mvar* = CON [(*row*,*col*)]

mvar: matrix variable name.
row: number of rows in matrix.
col: number of columns in matrix.

Purpose:

To set the value of each element in a matrix to one.

Remarks:

1. The form MAT *mvar* = CON is used for previously dimensioned matrices.
2. The form MAT *mvar* = CON (*row*,*col*) is used if the matrix was not previously dimensioned or if the matrix is to be redimensioned.
3. The matrix elements are set to ones regardless of any previously assigned values.

Example:

```
*LIST
0005 TAB =5
0010 DIM A[2,5]
0020 READ A[1,1],A[1,2],A[1,5]
0030 DATA 8,9,10,11,12
0040 MAT PRINT A
0050 MAT A=CON[2,4]
0060 MAT PRINT A

*RUN

 8     9     0     0     10
 0     0     0     0     0

 1     1     1     1          ←Matrix A after
 1     1     1     1              line 50.

END AT 0060
*
```

In line 50, matrix A is redimensioned and all elements of the matrix are assigned a value of one.

Identity Matrix (IDN)

MAT *mvar* = IDN [(*row*,*col*)]

mvar: matrix variable name.
row: number of rows in matrix.
col: number of columns in matrix.

S	✓
C	✓
F	

Purpose:

To set the elements of the major diagonal of the matrix to ones and the remaining elements of the matrix to zeros.

Remarks:

1. The major diagonal is defined as the diagonal that starts at the last element of the array and runs diagonally upward until the first row or first column is encountered.
2. The form MAT *mvar* = IDN is used for previously dimensioned matrices.
3. The form MAT *mvar* = IDN (*row*,*col*) is used if the matrix was not previously dimensioned or if the matrix is to be redimensioned.

Examples:

1.
```
*LIST
0025 TAB =5
0050 DIM A[4,4]
0100 MAT A=IDN
0150 MAT PRINT A

*RUN

 1    0    0    0
 0    1    0    0
 0    0    1    0
 0    0    0    1

END AT 0150
*
```

Identity Matrix
(IDN)

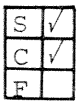
Examples:
(Continued)

2.
```
*LIST
0005 TAB =5
0010 DIM B[4,3]
0015 MAT PRINT B
0020 MAT B=IDN[2,3]
0025 MAT PRINT B

*RUN

 0     0     0
 0     0     0
 0     0     0
 0     0     0

 0     1     0          <-Matrix B after
 0     0     1              line 20.

END AT 0025
*
```

MATRIX I/O STATEMENTS

In addition to the matrix READ, INPUT and PRINT statements described in this section, there are several matrix file input/output statements which are described in Chapter 7.

MAT READ



MAT READ *mvar*[(*row*,*col*)] [,*mvar*[(*row*,*col*)]]...

mvar: matrix variable name.
row: number of rows in matrix.
col: number of columns in matrix.

Purpose:

To read values from the data list and assign them to the elements of the matrix or matrices listed in the MAT READ statement.

Remarks:

If a matrix was not previously dimensioned, it may be dimensioned in the MAT READ statement.

Example:

```
*LIST
0005 TAB =6
0010 MAT READ M[5,6]
0020 DATA 0,2,4,6,8,10,-9,-8,-7,-6,-5
0030 DATA -4,-3,-2,-1,0,1,3,5,7,9,11
0040 DATA .1,0,.5,7,-8,15,-15,35,41,13,18
0050 MAT PRINT M

*RUN

 0      2      4      6      8      10
-9     -8     -7     -6     -5     -4
-3     -2     -1      0      1      3
 5      7      9      11     .1     0
 .5     7     -8      15    -15     35

END AT 0050
*
```

Values from the data list are read into the 30-element matrix dimensioned as 5 x 6 in the MAT READ statement.

MAT INPUT

MAT INPUT *mvar*[(*row*,*col*)][,*mvar*[(*row*,*col*)]]...[;]

S	✓
C	✓
F	

mvar: matrix variable name.
row: number of rows in matrix.
col: number of columns in matrix.

Purpose:

To read values from the keyboard and assign the values to elements of a matrix or list of matrices when the program is run.

Remarks:

A matrix not previously dimensioned may be dimensioned in the MAT INPUT statement.

Data values, separated by either a comma or a carriage return, are entered for each element of the matrix. The list is terminated by a carriage return.

If the user does not supply enough data to fill the matrix before typing the carriage return, the program will continue to request data until each element of the matrix has been filled.

The data list may be terminated by a semi-colon, which leaves the cursor following the last input data item.

Example:

```
*LIST
0005 TAB =10
0010 MAT INPUT X[2,3]
0015 PRINT
0020 MAT PRINT X

*RUN
? 2,4,6? 77,7,9

 2            4            6
 77           7            9

END AT 0020
*
```

MAT PRINT

MAT PRINT *mvar* [{,;} *mvar*] ... [;]

mvar: matrix variable name.

S	✓
C	✓
F	

Purpose:

To output the values of the elements of a matrix or list of matrices to the user's terminal.

Remarks:

A matrix must be dimensioned by a DIM or other matrix statement before its use in the MAT PRINT statement.

If a semi-colon is used after a matrix variable in a MAT PRINT statement rather than a comma or carriage return it indicates that the matrix which immediately precedes the semi-colon is printed in compact format rather than zone format.

Example:

```
*LIST
0005 TAB =10
0010 DIM A[10,10]
0020 READ N
0030 MAT A=CON[N,N]
0050 FOR I=1 TO N
0060   FOR J=1 TO N
0070     LET A[I,J]=1/(I+J-1)
0080   NEXT J
0090 NEXT I
0130 MAT PRINT A
0190 DATA 4

*RUN

 1           .5          .333333     .25
 .5          .333333     .25         .2
 .333333     .25         .2          .166667
 .25         .2          .166667     .142857

END AT 0190
*
```

MATRIX CALCULATION
STATEMENTS

Addition and Subtraction

MAT *mvar*1 = *mvar*2 $\left\{\begin{matrix}+\\-\end{matrix}\right\}$ *mvar*3

mvar: matrix variable name.

S	✓
C	✓
F	

Purpose:

To perform the scalar addition or subtraction of two matrices.

Remarks:

1. Matrices *mvar*2 and *mvar*3 must have the same dimensions.
2. Matrix *mvar*1 may appear on both sides of the equal sign.
3. Arithmetic is performed on an element-by-element basis of *mvar*2 and *mvar*3 with the result assigned to the element of *mvar*1.

Example:

(Continued on next page)

Addition and
Subtraction

Example:
(Continued)

```
*LIST
0005 TAB =10
0010 DIM A[3,2],B[3,2],C[3,2]
0040 MAT READ B,C
0050 MAT A=B+C
0060 DATA -2,-5,3,4,.5,.1,6,4,-2,15,1.5,4
0070 MAT PRINT B
0080 MAT PRINT C
0090 MAT PRINT A

*RUN

-2          -5
 3           4
 .5          .1

 6           4
-2           15
 1.5         4

 4          -1
 1           19
 2           4.1

END AT 0090
*
```

Multiplication

MAT *mvar*1 = {*mvar*2 / (*expr*)} * *mvar*3

expr: any numeric expression enclosed in parentheses.
mvar: matrix variable names.

S	√
C	√
F	

Purpose:

To perform multiplication of a matrix by a numeric expression or another matrix.

Remarks:

1. Matrix *mvar*1 and *mvar*3 may represent the same matrix.
2. If two matrices (*mvar*2 and *mvar*3) are multiplied, the number of columns of *mvar*2 must equal the number of rows of *mvar*3. The resultant matrix (*mvar*1) will have the same number of columns as *mvar*3.
3. If a matrix is multiplied by a numeric expression, a scalar multiplication is performed on each element of the matrix.
4. To obtain the product of two matrices (*mvar*2 * *mvar*3), each row of *mvar*2 is multiplied by each column of *mvar*3. Each row/column set is added together to provide the resultant value of the matrix element in *mvar*1.

Examples:

(Continued on next page)

Examples:
(Continued)

1.
```
*LIST
0001 REM -  SCALAR MATRIX MULTIPLICATION
0005 TAB =10
0010 DIM A[2,2],B[2,2]
0020 MAT READ B
0030 MAT A=(5)*B
0040 DATA -.5,.8,1.5,-1
0050 MAT PRINT B
0060 MAT PRINT A

*RUN

-.5          .8
 1.5        -1

-2.5          4
 7.5         -5

END AT 0060
*
```

2.
```
*LIST
0001 REM - PRODUCT OF TWO MATRICES
0005 TAB =10
0010 DIM A[3,2],B[3,2],C[2,2]
0020 MAT READ B,C
0030 MAT PRINT B
0040 MAT PRINT C
0050 MAT A=B*C
0060 MAT PRINT A
0070 DATA 2,3,1.5,0,4,-1,-2,7,8

*RUN

 2            3
 1            5
 0            4

-1           -2
 7            8

 19           20
 34           38
 28           32

END AT 0070
*
```

Multiplication

Examples:
(Continued)

Matrix A is calculated as follows:

```
[B(1,1)*C(1,1)+B(1,2)*C(2,1)]   [B(1,1)*C(1,2)+B(1,2)*C(2,2)]
[B(2,1)*C(1,1)+B(2,2)*C(2,1)]   [B(2,1)*C(1,2)+B(2,2)*C(2,2)]
[B(3,1)*C(1,1)+B(3,2)*C(2,1)]   [B(3,1)*C(1,2)+B(3,2)*C(2,2)]
```

```
  [2*(-1)+3*7]        [2*(-2)+3*8]          19    20
= [1*(-1)+5*7]        [1*(-2)+5*8]    =     34    38
  [0*(-1)+4*7]        [0*(-2)+4*8]          28    32
```

Inverse Matrix (INV)

MAT *mvar*1 = INV (*mvar*2)

S	✓
C	✓
F	

mvar: matrix variable name.

Purpose:

To provide a matrix inversion of *mvar*2 and assign the resultant matrix element value to *mvar*1.

Remarks:

1. An inverse matrix is defined such that the product of a matrix and the inverse of the matrix is the identity matrix.
2. Matrix *mvar*2 must be a square matrix (at least 2 x 2).
3. Matrices may be inverted into themselves (i.e., *mvar*1 = *mvar*2 in the matrix INV statement).
4. The arithmetic of matrix inversion requires a knowledge of matrix determinants and of matrix cofactors. Determinants and cofactors for 2 x 2 matrices will be described here. For larger matrices, consult a mathematics text.

Matrix Determinants

Typically, the determinant of a 2 x 2 matrix can be obtained by multiplying along the diagonals and subtracting the second diagonal from the major diagonal.

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = (1*4) - (2*3) = -2$$

$$\begin{vmatrix} 1 & 5 \\ 3 & 20 \end{vmatrix} = (1*20) - (5*3) = 5$$

Inverse Matrix
(INV)

Remarks :
(Continued)

Matrix Cofactors

Cofactors of matrix elements for a 2 x 2 matrix are obtained by:

1. Reversing the elements along the major diagonal.
2. Changing the signs of the elements along the other diagonal.

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = \text{matrix A}$$

$$\begin{vmatrix} 4 & -2 \\ -3 & 1 \end{vmatrix} = \text{cofactors of matrix A}$$

Calculation of an Inverse Matrix

To obtain an inverse matrix, scalar multiply the cofactors of the matrix by the fraction (1/matrix determinant).

Example:

```
*LIST
0005 TAB =10
0010 DIM A[2,2]
0015 MAT READ A
0020 DATA 1,2,3,4
0030 MAT A=INV(A)
0040 MAT PRINT A

*RUN

-2            1
 1.5         -.5

END AT 0040
*
```

Inverse Matrix
(INV)

Example:
(Continued)

This example may be analyzed as follows:

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = \text{matrix A}$$

then:

$$\begin{vmatrix} 4 & -2 \\ -3 & 1 \end{vmatrix} = \text{cofactors of matrix A}$$

$$\begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} = (1*4) - (2*3) = -2 = \text{determinant of matrix A}$$

$$\text{INV (A)} = (1/-2) \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} = \begin{vmatrix} -2 & 1 \\ 1.5 & -.5 \end{vmatrix}$$

X Does not exist

Matrix Determinant (DET)

var = DET (X)

var: numeric variable.
X: dummy argument.

S	✓
C	✓
F	

Purpose:

To obtain the determinant of the last matrix inverted by an INV statement.

Remarks:

The value of the determinant calculated for the matrix is assigned to numeric variable *var*.

Example:

```
*LIST
0010 TAB =10
0020 DIM A[2,2]
0030 MAT READ A
0040 DATA 1,2,3,4
0050 MAT PRINT A
0080 MAT A=INV(A)
0090 MAT PRINT A
0100 LET B=DET(X)
0120 PRINT
0130 PRINT "DETERMINANT=";B

*RUN

 1            2
 3            4

-2            1
 1.5         -.5

DETERMINANT=-2

END AT 0130
*
```

Matrix Transposition (TRN)

MAT *mvar*1 = TRN (*mvar*2)

mvar: matrix variable name.

S	√
C	√
F	

Purpose:

To transpose matrix *mvar*2 and assign the resultant element values to *mvar*1.

Remarks:

1. A matrix is transposed by reversing the row and column assignments of the matrix elements.
2. A matrix cannot be transposed into itself.
3. The resultant matrix, *mvar*1, is redimensioned to the reversed row and column dimensions.

Example:

(Continued on next page)

Matrix Transposition
(TRN)

Example:
(Continued)

```
*LIST
0010 TAB =10
0020 DIM B[3,4]
0030 MAT READ B
0040 DATA 4,5,7,9,0,0,0,0,1,3,5,7
0050 MAT PRINT B
0060 PRINT
0070 PRINT
0080 MAT A=TRN(B)
0090 MAT PRINT A

*RUN

 4          5          7          9
 0          0          0          0
 1          3          5          7


 4          0          1
 5          0          3
 7          0          5
 9          0          7

END AT 0090
*
```

Notice that B(1,2) is equal to A(2,1).

CHAPTER 7

FILE INPUT AND OUTPUT

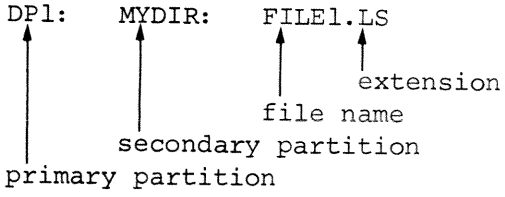
FILE CONCEPTS

Definition of a File

The user files referred to in this manual are those which have been created on an RDOS system. A user "filename" is specifically defined as follows:

[primary part.:][secondary part.:][sub-dir.:]file name [.ex]

For example:



Briefly, a file is a collection of information that is known by, and accessible by, a "filename" which may be a reserved device (e.g., $CDR) or a file stored on disk.

In BASIC, a random access file is one in which individual records in the file can be accessed for reading or writing. A BASIC random access file should not be confused with a randomly organized RDOS file. BASIC random access files may be RDOS random, sequential or contiguous files.

File Name and Extension

"Filenames" may be written as string literals or as string variables in BASIC.

File Name and Extension (Continued)

The name of the file in the "filename" must conform to RDOS requirements for extended file names. Therefore, a file name may consist of as many as ten characters (26 alphabetic, 10 numeric, and the dollar sign ($) character) plus an optional two character alphanumeric extension, separated from this file name by a period (.).

For example:

TEST.SR	is a meaningful way to signify a source file.
TEST.CI	could signify the core image file obtained by SAVEing TEST.SR.
TEST.LS	could be a listing file output from the program.

Unlike RDOS utility programs such as MAC and RLDR, BASIC does not recognize any special extensions such .SR, .SV, .LS, etc. Extensions may be constructed to suit the programmer's needs.

Reserved File Names

Unit record devices and magnetic tape devices are given special names and do not have extensions. Devices with reserved names are listed below:

$CDR and $CDR1	Punched card readers
CTn	Cassette units ($0 \leq n \leq 17_8$)
$LPT and $LPT1	Line printers
MTn	Magnetic tape units ($0 \leq n \leq 17_8$)
$PLT and $PLT1	Incremental plotters (access via assembly language subroutines)
$PTP and $PTP1	Paper tape punches
$PTR and $PTR1	Paper tape readers

File Attributes

A number of file attributes may be specified which permit such features as file sharing, read and write protection, etc. The file attributes are changed, added or removed by use of the CHATR command described in Chapter 8.

OPEN FILE

OPEN FILE(*file*,*mode*),*filename*[{,*record size*}/{,}[,*file size*]]

S	✓
C	✓
F	

file: a numeric expression which evaluates to a file number in the range 0 to 7. The file number is associated with *filename* and is used for further references in other file I/O statements.

mode: a numeric expression which evaluates to a number in the range 0 to 6 and is used to specify the manner in which a file is to be accessed. The modes are described under Remarks.

filename: a string literal or string variable constructed in a manner previously described in this chapter which evaluates to the name of a file.

record size: an optional numeric expression which evaluates to the fixed length (in bytes) of each record in a random or contiguously organized file and is applicable to modes Ø and 4 through 6 only. *Record size* may be any value from 1 to 32768 and if not specified, a default value of 128 bytes per record is assigned.

file size: an optional numeric expression which evaluates to the number of records when creating a contiguously organized file and thereby establishes a limit for its size.

OPEN FILE
(Continued)

Purpose:

To link a filename or system device with a file number for further referencing in file I/O statements.

Remarks:

1. For maximum efficiency it is recommended that fixed length record modes of operation be used whenever possible (modes Ø and 4 through 6). Record lengths should be specified as closely as possible to the length of data actually written or read from the file.

   Record length may be calculated as follows:

   •Numeric Data
   Single Precision - 4 bytes per data item
   Double Precision - 8 bytes per data item

   •String Data
   one byte per character in string +1 (for null character)

   •Arrays
   (No. of rows) * (No. of columns) * (precision (4 or 8))

2. Modes 0 to 6 are described as follows:

   Mode 0 - Random access file (for input and/or output). Only disk files may be opened in random mode for reading and writing. Record length is fixed by *record size* or by the default value. If no disk file having the *filename* specified in the OPEN FILE statement is found in the user's directory, a new disk file is created and an entry is made for *filename* in the directory.

   Mode 1 - Output (write to a new file). Either a disk file or an appropriate output device can be opened in this mode. Records may be variable in length. Only writes are permitted to the file. If a file of this name already exists in the user's directory,

Remarks:
(Continued)

the previous copy is first deleted from the disk. In either case, a new file is created (initialized with 0 length).

Mode 2 - Output (append to an existing file)
Any appropriate file may be opened in append mode. When opened, the file pointer is positioned to the end of the file so that subsequent data written to the file will extend it. If the file does not exist in the user's directory, an entry for the file name will be made in the directory and a new file is created. Records may be variable in length.

Mode 3 - Input (for reading only)
Either a disk file or appropriate input device can be opened in this mode. If a disk file is opened in this mode, the file must already exist. Only reads are permitted for a file opened in Mode 3. If the file is not found in the user's directory, a search for the file is made in the public directory. Records may be variable in length.

Modes 4,5,6 - Correspond to Modes 1, 2 and 3, respectively, in function but contain fixed length records rather than variable length records. Modes 4, 5 and 6 always read/write a fixed number of bytes equivalent to the *record size* specified in the most recent OPEN statement for the file. When the read/write is complete, the file pointer will automatically be moved ahead to the beginning of the next record if the number of bytes read/written is less than the *record size*.

Files that are created using Modes 0, 4, 5 and 6 may later be read/written. For example a file created in Mode 4 may be later opened in Mode 0, 5 or 6.

Remarks:
(Continued)

3. The following table summarizes the various combinations of arguments to the OPEN FILE statement and shows the resultant files created. Existing RDOS files may be OPENed in any BASIC mode.

	IF			THEN			
BASIC FILE TYPE	BASIC MODE	FILE EXIST?	FILE-SIZE SPECI-FIED?	DELETE OLD FILE	CREATE SEQ. FILE	CREATE RAND. FILE	CREATE CONT. FILE
BASIC RANDOMLY ACCESSED FILE (INPUT/ OUTPUT)	0	YES	YES	NO	NO	NO	NO
	0	YES	NO	NO	NO	NO	NO
	0	NO	YES	NO	NO	NO	YES
	0	NO	NO	NO	NO	YES	NO
BASIC SEQUEN-TIALLY ACCESSED FILE (OUTPUT)	1, 4	YES	YES	YES	NO	NO	YES
	1, 4	YES	NO	YES	YES	NO	NO
	1, 4	NO	YES	NO	NO	NO	YES
	1, 4	NO	NO	NO	YES	NO	NO
BASIC SEQUEN-TIALLY ACCESSED FILE (APPEND)	2, 5	YES	YES	NO	NO	NO	NO
	2, 5	YES	NO	NO	NO	NO	NO
	2, 5	NO	YES	NO	NO	NO	YES
	2, 5	NO	NO	NO	YES	NO	NO
BASIC SEQUEN-TIALLY ACCESSED FILE (INPUT)	3, 6	YES	YES	NO	NO	NO	NO
	3, 6	YES	NO	NO	NO	NO	NO
	3, 6	NO	YES	ERROR	ERROR	ERROR	ERROR
	3, 6	NO	NO	ERROR	ERROR	ERROR	ERROR

NOTE: CREATE'S above refer to RDOS organization types.

Examples:
(Continued)

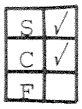
```
*100 OPEN FILE (1,4),"NETSAK.JR", 256,128
```

This statement opens file 1, named NETSAK.JR, as a contiguously organized output file with a record size of 256 bytes per record and a file size of 128 records.

```
*100 OPEN FILE (2,0),"RESSEHC.TO", 20
```

This statement opens the file named RESSEHC.TO as file number 2 for random access of its records which are 20 bytes long.

CLOSE FILE



CLOSE [FILE (*file*)]

file: a numeric expression which evaluates to a file number previously associated with a *filename* in an OPEN FILE statement.

Purpose:

To disassociate a *filename* and a file number so that the file can no longer be referenced.

Remarks:

1. The CLOSE FILE statement may be used to close a file so that it may be reopened by an OPEN FILE with a new mode argument.
2. The CLOSE form of the statement closes all open files.

Examples:

```
*100  CLOSE FILE (1)
*200  CLOSE FILE (X+3)
*300  CLOSE
```

WRITE FILE

WRITE FILE $\left(\left\{\begin{matrix} file \\ file, record \end{matrix}\right\}\right), \left\{\begin{matrix} expr \\ var \\ svar \\ "string\ lit" \end{matrix}\right\} \left[, \left\{\begin{matrix} expr \\ var \\ svar \\ "string\ lit" \end{matrix}\right\}\right] \ldots$

S	✓
C	✓
F	

file: a numeric expression which evaluates to the number of a file opened in Mode 0 for random access, or Mode 1, 2, 4 or 5 for sequential access.

record: a numeric expression which evaluates to the number of a record in a file opened for random access (Mode 0).

expr, *var*, *svar*, and *string lit:* a list of one or more numeric expressions, numeric variables, string variables, and literals whose values are written into a sequential access file or a record in a random access file.

Purpose: To write data in binary format into a sequential access file or a record in a random access file.

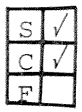
Remarks: The number of the first record in a random access file is zero (0).

Example: (Continued on next page)

Example:
(Continued)

```
*LIST
0001 REM-FILE WRITE
0005 TAB =10
0010 DIM A[3,4]
0020 FOR I=1 TO 3
0030   FOR J=1 TO 4
0040     LET A[I,J]=((I-1)*4+J)*3
0050   NEXT J
0060 NEXT I
0070 MAT PRINT A
0080 PRINT
0090 OPEN FILE[1,0],"TESTFILE",20
0100 FOR I1=1 TO 3
0110   LET I=4-I1
0120   FOR J1=1 TO 4
0130     LET J=5-J1
0140     LET R=(3-I)*4+(5-J)
0150     WRITE FILE[1,R],A[I,J]
0160     PRINT A[I,J],
0170   NEXT J1
0180   PRINT
0190 NEXT I1
0200 CLOSE

*RUN

 3            6            9            12
 15           18           21           24
 27           30           33           36

 36           33           30           27
 24           21           18           15
 12           9            6            3

END AT 0200
*
```

READ FILE



READ FILE $\left(\left\{\begin{matrix}file\\file,record\end{matrix}\right\}\right), \left\{\begin{matrix}var\\svar\end{matrix}\right\} \left[, \left\{\begin{matrix}var\\svar\end{matrix}\right\}\right] \ldots$

file: a numeric expression which evaluates to the number of a file opened in Mode 0 for random access, or Mode 3 or 6 for sequential access.

record: a numeric expression which evaluates to the number of a record in a file opened for random access (Mode 0).

var and *svar:* a list of one or more numeric variables and string variables which are assigned values read sequentially from a randomly accessed record (Mode 0) or sequentially from a file (Mode 3 or 6).

Purpose:

To read data in binary format from a sequentially accessed file or from the records of a randomly accessed file.

Remarks:

1. Each numeric variable or string variable in the READ FILE variable list must correspond in data type to the corresponding data item being read from the file or record within the file.
2. The number of the first record in a random access file is zero (0).
3. In random access files, records which have not been written into will contain all zeros when read.
4. The EOF function may be used to detect an end-of-file on the file which is being read.

Example:

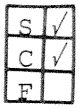
(Continued on next page)

Example:
(Continued)

```
*LIST
0001 REM-READ FILE
0005 TAB =10
0010 DIM B[3,4]
0020 OPEN FILE[1,0],"TESTFILE",20
0030 FOR I=1 TO 12
0040   LET I1=INT((I-1)/4)+1
0050   LET J1=I-(4*(I1-1))
0060   READ FILE[1,I],B[I1,J1]
0070 NEXT I
0080 MAT PRINT B
0090 CLOSE

*RUN

 36          33          30          27
 24          21          18          15
 12          9           6           3

END AT 0090
*
```

Note: This program uses the file TESTFILE which is created in the program example provided with the WRITE FILE statement.

PRINT FILE



PRINT FILE (*file*) { *expr* / *var* / *svar* / *"string lit"* } [{ , / ; } { *expr* / *var* / *svar* / *"string lit"* }] ... [{ , / ; }]

file: a numeric expression which evaluates to the number of a file opened in Mode 1 or 2 for sequential output.

expr, var, svar and *string lit:* a list of one or more numeric expressions, numeric variables, string variables, and string literals whose values are written into a sequential access file.

Purpose:

To write data in ASCII into a sequential access file.

Remarks:

1. This statement is intended for outputting to an ASCII device such as a line printer, or to a disk file for later off-line printing.
2. Each item in the expression list must be separated from the next by a comma, semicolon, or carriage return. Output formatting is identical to that discussed in Remarks for the PRINT statement.

Example:

```
*10  OPEN FILE(3,1),"$LPT"
*100 PRINT FILE(3),"OUT6"
*200 PRINT FILE(3),"X=";X,"XSQR=";X↑2,"XCUBE";X↑3
```

INPUT FILE

S	√
C	√
F	

INPUT FILE (*file*) $\left\{\begin{matrix} var \\ svar \end{matrix}\right\}$ $\left[,\left\{\begin{matrix} var \\ svar \end{matrix}\right\}\right]\ldots$

file: a numeric expression which evaluates to the number of a file opened in Mode 3 for sequential access.

var and *svar:* a list of one or more numeric variables and string variables whose values are read from a sequential access file.

Purpose :

To read data in ASCII from a sequential access file.

Remarks:

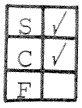
1. Each numeric variable or string variable in the INPUT FILE variable list must correspond in data type to the corresponding data item being read from the file.
2. The data file must be formatted such that commas or carriage returns are used to separate data items.

Example:

```
*40 OPEN FILE (1,3), "$PTR"
*70 INPUT FILE (1), Z,Y X,A$,B$
```

MAT WRITE FILE

MAT WRITE FILE $\left(\left\{\begin{matrix} file \\ file, record \end{matrix}\right\}\right)$, $mvar\left[, mvar\right]\ldots$



file: a numeric expression which evaluates to the number of a file opened in Mode 0 for random access, or Mode 1, 2, 4 or 5 for sequential access.

record: a numeric expression which evaluates to the number of a record in a file opened for random access (Mode 0).

mvar: a list of one or more matrices whose values are written into a record (Mode 0) or a file (Mode 1, 2, 4 or 5).

Purpose:

To write matrix data in binary format into a sequential access file or a record in a random access file.

Remarks:

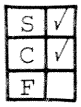
1. Matrix arrays listed in the MAT WRITE FILE statement must be previously dimensioned.
2. The number of the first record in a random access file is zero (0).
3. Matrices written in Modes 4 and 5 must fit into a record whose length is specified in the OPEN statement for the corresponding file.

Example:

```
*50 OPEN FILE (0,1),"AAA"
*80 MAT WRITE FILE (0),B,C,X
```

MAT READ FILE

MAT READ FILE $\left(\left\{\begin{matrix} file \\ file, record \end{matrix}\right\}\right)$, $mvar\left[, mvar\right]\ldots$



file: a numeric expression which evaluates to the number of a file opened in Mode 0 for random access, or Mode 3 or 6 for sequential access.

record: a numeric expression which evaluates to the number of a record in a file opened for random access (Mode 0).

mvar: a list of one or more matrices which are assigned values read sequentially from a randomly accessed record (Mode 0) or sequentially from a file (Mode 3 or 6).

Purpose:

To read data in binary format, for the elements of matrix arrays, from a sequentially accessed file or from the records of a randomly accessed file created by MAT WRITE FILE statements.

Remarks:

1. Previously dimensioned matrix arrays may be listed in the statement by name only. Matrix arrays which have not been dimensioned must be dimensioned in the MAT READ FILE statement.
2. In random access files, records which have not been written into will contain all zeros when read.
3. Data items are read from the file, or record, sequentially and are assigned to the array elements by row.
4. The number of the first record in a random access file is zero (0).
5. The EOF function may be used to detect an end-of-file on the file which is being read.
6. The amount of data to be read must not exceed the *record size* specification for files OPENed in Modes Ø or 6.

MAT READ FILE
(Continued)

Examples:

```
*10 DIM A(7,3), B(12,7)
*30 OPEN FILE (1,3),"MATRIXA"
*40 MAT READ FILE (1), A,B,C(3,4),D(5)
```

MAT PRINT FILE

MAT PRINT FILE(*file*),*mvar* $\left\{\begin{matrix},\\;\end{matrix}\right\}$*mvar* ... $\left[\left\{\begin{matrix},\\;\end{matrix}\right\}\right]$

S	✓
C	✓
F	

file: a numeric expression which evaluates to the number of a file opened in Mode 1 or 2 for sequential output.

mvar: a list of one or more matrices whose values are written to a sequential access file.

Purpose:

To write matrix data in ASCII into a sequential access file.

Remarks:

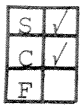
1. This statement is intended for outputting to an ASCII device such as a line printer, or to a disk file for off-line printing.
2. The MAT INPUT FILE statement cannot be used to input data which was output by MAT PRINT FILE because the MAT PRINT FILE statement does not output delimiters between matrix elements.
3. If a semi-colon is used after a matrix variable in the MAT PRINT FILE statement rather than a comma or carriage return it indicates that the matrix which immediately precedes the semi-colon is printed in compact format rather than zone format.

Example:

```
*5  DIM B(20,20)
*10 OPEN FILE (0,1),"Z.22"
*20 MAT PRINT FILE (0),B
```

MAT INPUT FILE

MAT INPUT FILE (*file*),*mvar* [,*mvar*]...



file: a numeric expression which evaluates to the number of a file opened in Mode 3 for sequential access.

mvar: a list of one or matrix arrays whose values are read from a sequential access file.

Purpose:

To read matrix data in ASCII from a sequential access file.

Remarks:

1. Previously dimensioned matrix arrays may be listed in the statement by name only. Matrix arrays which have not been dimensioned must be dimensioned in the MAT INPUT FILE statement.
2. Data items are read from the file sequentially and are assigned to the array elements by row.
3. The EOF function may be used to detect an end-of-file on the file which is being read.

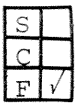
Example:

```
* 5 DIM Y(7,6),Z(13,2)
*10 OPEN FILE (2,3), "XX.AA"
*50 MAT INPUT FILE (2),X(5,5),Y,Z
```

EOF (X)

EOF (*file*)

file: a numeric expression which evaluates to the number of a file opened for reading in Mode 0, 3 or 6.



Purpose:

To detect the end of data when transferring data from a file.

Remarks:

1. The EOF function returns an integer indicating whether or not the last READ from *file* included an end-of-file delimiter.
2. If an end-of-file was detected, the function returns a value of +1; otherwise the function returns a 0.
3. When the EOF function is used in conjunction with the IF-THEN statement, a conditional transfer can be made if an end-of-file is detected.
4. Random files (Mode 0) return an EOF if the user attempts to read a record number larger than the last written in the file. The file must be closed and reopened to continue.

Example:

```
*100 OPEN FILE (1,3), "$PTR"
*110 READ FILE (1), A,B,C,D,E
*120 PRINT A,B,C,D,E
*130 IF EOF (1) GOTO 200
*140 GOTO 110
*200 CLOSE FILE (1)
```

CHAPTER 8

INTERACTIVE SYSTEM COMMANDS

INTRODUCTION

The preceding chapters have described the statements and functions used for writing programs in the BASIC language. However, Extended BASIC may also be used interactively to perform the following functions:

- Maintain BASIC source programs
- Maintain disk directories
- Dynamically debug programs
- Perform desk calculator functions
- Communicate with the system operator and other users.

The commands necessary to perform these functions are described in this chapter.

NEW

NEW

S	✓
C	✓
F	

Purpose:

To delete the currently stored program statements and variables, and to close any open files.

Remarks:

1. The programmer's storage area must be cleared with a NEW command (or statement) before entering a new program to avoid lines from previous programs being executed along with the new program.
2. The NEW statement can be the last executable statement within a program thereby clearing the program from memory after program execution is completed.
3. When used with the ON ESC or ON ERR statements, the NEW statement can be used to prevent unauthorized access to a program.

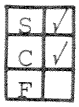
Example:

```
*LIST
0100 READ A,B,C,D
0110 LET E=A*23
0115 LET F=C*A
0120 PRINT E;F
0130 NEW
0135 DATA 1,2,3,4

*RUN
 23   3

*LIST
ERROR 13 - LINE NUMBER
*
```

Does not work

ERASE

ERASE *line n1*, *line n2*



line n1 and *line n2*: line numbers in a program.

Purpose:

To remove statements from a program.

Remarks:

1. This command may be used to remove *line n1* through *line n2*, inclusively, in the user's program. This command simplifies the editorial process of deleting only one line at a time.
2. Typically, this command might be used to clear an area in a program to permit a subsequent ENTER of a program whose lines are in the same range as those deleted.
3. If no lines exist in the user's program in the range *line n1* to *line n2*, then an error message is output to the user's terminal.

Example:

```
ERASE 1500, 1900)      ←Delete lines 1500 through
                         1900 inclusive.
```

LIST

S		
C	√	
F		

LIST [*line n1* | TO *line n2* | *line n1* { TO / , } *line n2*] [*filename*]

line n1: first statement to be listed.
line n2: last statement to be listed.
filename: a device or disk file expressed as a string literal.

Purpose:

To output part or all of the current program in ASCII to the device specified by *filename* or to the terminal if *filename* is not specified.

Remarks:

1. The variations of the LIST command are described as follows:

LIST)	- List the entire program starting at the lowest numbered statement.
LIST *n1*)	- List only the single statement at line number *n1*.
LIST TO *n2*)	- List from the lowest numbered line through line number *n2*, inclusive.
LIST *n1* { TO / , } *n2*)	- List from line numbers *n1* through line number *n2*, inclusive.

2. When the *filename* argument is included, the LIST command causes the specified lines to be written to a file called *filename*, or to the device called *filename*.

3. The file created by the LIST command can be read back into the program storage area by the ENTER command. If statements are listed to a disk file, *filename* is entered in the programmer's directory, replacing any previous file of the same name.

LIST
(Continued)

Examples:

*LIST 700,9999)	Line numbers 700 through 9999 are listed at the terminal.
*LIST "$LPT")	The entire program is output to the line printer.
*LIST 20)	List line number 20 at the terminal.
*LIST "TEST.SR")	The current program is output to the programmer's directory in ASCII with the filename TEST.SR and replaces any previous file with that name.

PUNCH

S	
C	✓
F	

PUNCH [*line n1* [TO *line n2* | *line n1* { TO / , } *line n2*]]

line n1: first statement to be punched.
line n2: last statement to be punched.

Purpose:

To output part or all of the current program in ASCII to the terminal punch.

Remarks:

1. A leader of null characters precedes the punched listing and a trailer of null characters follows the listing.

2. The number of null characters punched as leader and trailer is equivalent to the number of characters defined as the page width (see PAGE command). This represents 13.2 inches of leader for a 132 character line.

3. The PUNCH command does not turn on the terminal punch. The following procedure is required:

   a. Type the desired PUNCH command followed by a carriage return and immediately press the ON button on the terminal punch.
   b. A null leader will be punched, followed by a listing of the desired lines of the current program, followed by a null trailer.
   c. When punching is completed, press the OFF button on the punch.

4. The variations of the PUNCH command are described as follows:

   PUNCH) - Punch the entire program starting at the lowest numbered statement.

PUNCH

Remarks:
(Continued)

PUNCH $n1$	- Punch only the single statement at line number $n1$.
PUNCH TO $n2$	- Punch from the lowest numbered line to line number $n2$, inclusive.
PUNCH $n1$ {TO, ,} $n2$	- Punch from line number $n1$ through line number $n2$, inclusive.

Example:

```
*PUNCH 200 TO 500
```

Punch line numbers 200 through 500 of the current program.

SAVE

SAVE *filename*

filename: The name of a disk file or a device.

S	√
C	√
F	

Purpose:

To write the current program in binary format to the device or disk file named by *filename*.

Remarks:

1. If *filename* is a disk file, then *filename* is entered into the programmer's directory, replacing any file of the same name.
2. A SAVEd program can be LOADed, CHAINed, or RUN.
3. A SAVEd program which is LOADed can then be LISTed in ASCII format.
4. SAVEing a program in binary format is more efficient than LISTing in ASCII when storing a program.
5. A SAVEd program may not run under all configurations of BASIC. In particular, if the precision of the floating point representation in the RUN environment is different from that of the SAVE environment, the program will not even be loadable.

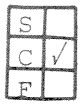
Example:

```
*SAVE "FA.BC"
*SAVE "$PTP"
*SAVE S$ (1,7)
```
} Commands

```
*10  SAVE"OURSHIP"
*20  SAVE B$
```
} Statements

LOAD

LOAD *filename*



filename: the name of a binary file created by a previous SAVE command.

Purpose:

To load a previously SAVEd program in binary format into the program storage area.

Remarks:

1. The LOAD command executes an implicit NEW command (clearing the storage area) and then reads *filename* into core.
2. *Filename* may be on disk or may be on a binary input device such as the paper tape reader.
3. If *filename* is a disk file, a search is made for *filename* in the programmer's directory first. If not found, a search is made in the library directory for *filename*.
4. When a *filename* is LOADed, it can be LISTed, modified, or RUN.

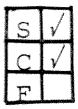
Example:

```
*LOAD "$PTR")
*LOAD "MATH3")
*LOAD "MTØ:1")
```

ENTER

ENTER *filename*

filename: a device or disk file.



Purpose:

To merge the BASIC statement lines from the device or disk file named by *filename* into the programmer's current program storage area.

Remarks:

When statement lines from an ENTERed *filename* have the same statement numbers as lines in the current program, the current program statement lines are replaced.

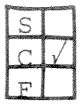
Example:

```
*NEW)
*ENTER "TEST1.SR")
*ENTER "TEST2.SR")
*LIST "FINAL.SR")
```

The programmer's storage area is cleared and source programs TEST1.SR and TEST2.SR are merged with the resultant program stored in the programmer's directory as FINAL.SR.

RUN

RUN [*line no.* / *filename*]



line no.: the line in the current program from which execution is to begin.

filename: the name of a disk file or device.

Purpose:

To execute a program either from the first line number in the program or from a specified line number in the program.

Remarks:

The variations of the RUN command are described as follows:

RUN) — Clear all variables, undimension all arrays and strings, do a RESTORE, initialize the random number generator, and then run the current program from the first line number.

RUN n) — All existing information (variable values, dimensioning, etc.) resulting from a previous execution of the current program are retained and the current program is run starting at the line numbered n. This form of the RUN command allows resumption of program execution retaining current values of all variables and parameters. It may be used after a STOP or after an error and will incorporate any alterations to the program that may have been made after the STOP or error occurred.

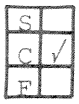
Remarks:
(Continued)

RUN "filename"↵	If the file is on disk, the system follows the search procedure outlined in the LOAD command. When *filename* is found, the command executes a NEW, clearing the current program area, executes a LOAD, and then executes the new current program.

Examples:

```
*RUN↵
*RUN "$PTR"↵
*RUN 250↵
*RUN "MATH3"↵
*RUN "MT1:0"↵
```

RENUMBER



RENUMBER [{*line n1* | STEP *line n2* | *line n1* STEP *line n2*}]

line n1: the initial line number for the current program.

line n2: the increment between line numbers for the current program.

Purpose:

To renumber the statements in the current program.

Remarks:

1. The variations of the RENUMBER command are described as follows:

RENUMBER)	Renumber the current program starting with default line number 0010 with a default increment of 10 between line numbers.
RENUMBER *n1*)	Renumber the current program starting with line number *n1* and incrementing by *n1* between line numbers.
RENUMBER STEP *n2*)	Renumber the current program starting with default line number 0010 and incrementing by *n2* between line numbers.
RENUMBER *n1* STEP *n2*)	Renumber the current program starting with line number *n1* and incrementing by *n2* between line numbers.

2. Line numbers are limited to a four-digit number. If a RENUMBER command causes a line number to be greater than 9999, the command is re-executed as:

RENUMBER 1 STEP 1

Remarks:
(Continued)

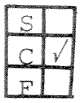
3. The RENUMBER command also modifies the line numbers in IF-THEN, GOTO, and GOSUB statements to agree with the new line numbers assigned to the current program.

4. Line numbers which cannot be resolved are changed to 0000 and an error message is issued.

Example:

```
*LIST
0005 TAB =5
0010 DIM A[3,4]
0020 LET A[1,2]=6
0030 LET A[3,4]=10
0040 MAT PRINT A
0050 MAT A=ZER[3,3]
0060 MAT PRINT A

*RENUMBER 10 STEP 5
*LIST
0010 TAB =5
0015 DIM A[3,4]
0020 LET A[1,2]=6
0025 LET A[3,4]=10
0030 MAT PRINT A
0035 MAT A=ZER[3,3]
0040 MAT PRINT A
```

CON



CON

Purpose:

To continue the execution of a program after a STOP statement in the program has been executed, the ESCape key has been pressed, or an error has occurred.

Remarks:

1. The CON command is equivalent to a RUN *line no.* command where *line no.* is equal to the statement directly following the statement at which the program stopped.
2. If a run-time error is encountered within the program, the user may correct the error and issue the CON command to begin execution from the statement where the error occurred.

Example:

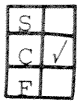
(Continued on next page)

CON
(Continued)

Example:

```
*LIST
0010 PRINT "PRINCIPAL   INT(%)    ";
0020 PRINT "TERM(YRS)   TOTAL"
0030 READ P,I,T
0035 IF T=0 THEN GOTO 0080
0040 LET A=P*(1+I/100)↑T
0050 PRINT P; TAB(12);I; TAB(21);T; TAB(32);A
0060 GOTO 0030
0070 DATA 1000,5,10,0,0,0
0080 PRINT
0090 PRINT "CHANGE DATA AT LINE 70"
0100 STOP
0110 GOTO 0010

*RUN
PRINCIPAL   INT(%)   TERM(YRS)   TOTAL
 1000         5         10         1628.9

CHANGE DATA AT LINE 70

STOP AT 0100
*70 DATA 2500,3,10,1450,6,12,0,0,0
*CON
PRINCIPAL   INT(%)   TERM(YRS)   TOTAL
 2500         3         10         3359.79
 1450         6         12         2917.7

CHANGE DATA AT LINE 70

STOP AT 0100
*
```

SIZE



SIZE

Purpose:

To print the number of bytes used by the program and the total number of bytes that are still available.

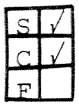
Example:

SIZE
USED: 9329 BYTES
LEFT: 8077 BYTES

Out of a total of 17,406 bytes of memory available for program and data storage, 9329 are occupied and 8077 remain.

BYE

BYE



Purpose:

To sign-off the system and make the terminal available to others.

Remarks :

1. BYE may be used as a keyboard command or as a program statement to automatically log the user off the system.
2. A display of accounting information precedes the sign-off.
3. Telephone connections are severed.
4. The ESC key will not be recognized once the BYE command is begun.

Example:

```
*BYE
      01/02/74  10:06  SIGN OFF, 04   (terminal no.)
      01/02/74  10:06  CPU USED, 206  (time in seconds)
      01/02/74  10:06  I/O USED, 11   (number of I/O's made)

DGC READY
```

PAGE

PAGE=*expr*

S	✓
C	✓
F	

expr: an arithmetic expression in the range: $1 \leq n \leq 132$.

Purpose: To set the right margin of the terminal.

Remarks: A default value of 72 is used as the maximum line width.

Example:

```
*LIST
0010 PAGE =30
0020 FOR I=1 TO 25
0030   PRINT I;
0040 NEXT I

*RUN
 1  2  3  4  5  6  7  8  9  10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25
END AT 0040
*
```

TAB

S	√
C	√
F	

TAB=*expr*

expr: an arithmetic expression in the range: $1 \leq n \leq$ page width given by PAGE command.

Purpose:

To set the zone spacing between the data output by PRINT statements.

Remarks:

1. The default zone spacing is 14 columns. This spacing allows five zones of output data per 72 character teletypewriter line.
2. Since the maximum range of zone spacing depends upon the PAGE command setting, it is good practice to set the page width first and then the zone spacing.

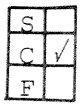
Example :

```
*LIST
0010 PAGE =50
0020 TAB =10
0030 FOR I=1 TO 25
0040   PRINT I,
0050 NEXT I

*RUN
 1          2          3          4          5
 6          7          8          9          10
 11         12         13         14         15
 16         17         18         19         20
 21         22         23         24         25

END AT 0050
*
```

MSG

MSG[Δ*userID*Δ*message*]

userID: identification of receiving user.
message: text of message.



Purpose:

To transmit a message from the programmer's terminal to any other programmer or to the operator.

Remarks:

1. The operators ID is:

   OPER

2. If a receiving programmer has set the message lockout command (NOMSG) or is not on line, then the transmission will not be successful and an error message will be printed at the sender's terminal.
3. If the transmission is successful, then the following is printed at the receiving programmer's terminal:

   FROM *sendersID: message*

   where *sendersID* is the identification of the programmer that sent the *message*.

4. Message length is limited to one line.
5. Quotation marks are not necessary for message.
6. A blank is necessary between MSG, *userID* and *message*.
7. When used without operands, the MSG command resets the action of a NOMSG command to enable the reception of messages.

MSG
(Continued)

(Continued)

Example:

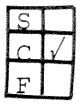
MSGΔOPERΔMOUNT MY CASSETTE-THANKS)

At master console:

FROM JACK: MOUNT MY CASSETTE-THANKS

NOMSG

NOMSG



Purpose:

To prevent reception of messages from other programmers.

Remarks:

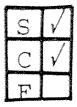
1. The system operator may override the NOMSG command for important messages.
2. The user can cancel the NOMSG command by using the MSG command with no operands.
3. NOMSG does not affect the programmer's ability to transmit messages.

Example:

```
*NOMSG)
*RUN"PROG.3")

END AT 300
*MSG)
```

NOESC

NOESC



Purpose:

To disable ESC key operation.

Remarks:

1. The NOESC statement, or command, can be used to prevent the interruption of a program which occurs when the ESC key is pressed.
2. If a programmer's log-on identification includes a log-on program which is executed, then a NOESC condition is invoked by default. The programmer can circumvent the NOESC condition by either including an ESC statement as the last statement in the log-on program or by typing an ESC command after log-on.
3. No action is taken if subsequent NOESC commands or statements are encountered without intervening ESC commands.

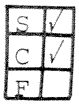
Examples:

```
*NOESC)          ←command

*10 NOESC        ←statement
```

Doc. not found

ESC

ESC



Purpose:

To re-enable ESC key operation.

Remarks:

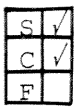
1. The ESC key can be disabled by a NOESC or by default at log-on if a log-on file is executed.
2. No action is taken if subsequent ESC commands or statements are encountered without intervening NOESC commands.
3. See NOESC for additional remarks.

Example:

```
*ESC)          ←command

*10 ESC        ←statement
```

NOECHO

NOECHO



Purpose:

To inhibit the echoing of input at the programmer's terminal.

Remarks:

1. NOECHO does not affect program execution or printed output.
2. NOECHO can be useful when entering sensitive data such as passwords.
3. NOECHO can be cancelled by ECHO.
4. No action is taken if subsequent NOECHO commands or statements are encountered without intervening ECHO commands.

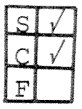
Example:

```
*NOECHO)          ←command

*10 NOECHO        ←statement
```

does not have

ECHO

ECHO



Purpose:

To re-enable echoing of input at the programmer's terminal.

Remarks:

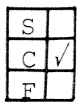
1. ECHO cancels a NOECHO command or statement.
2. BASIC takes no action if subsequent ECHO commands or statements are encountered without intervening NOECHO commands.
3. See NOECHO for additional remarks.

Example:

```
*ECHO)          ←command

*10 ECHO        ←statement
```

FILES

FILES



Purpose:

To print all file names in the programmer's directory.

Remarks:

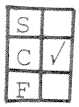
One file name is printed per print zone.

Example:

```
*FILES
157.          134.          GOSUB1.SR
STOP.SR       121.          NEW.
ON.ES         READ.SR       116.
TIME.         FOR1.SR       FOR2.SR
MORSE.        110.SR        FOR4.SR
113.          TAB.SR        132A.
COM.CM        CON.          110.
TAB.          SUBSTRINGS.   CONCAT.
115.          GOTO.         111A.
PAGE.         HELLO.SV      PRINT1.SR
PRINT2.SR     109B.         PRINT3.SR
PRINT4.SR     92.           INPUT2.SR
107.          117.          IF3.
*
```

LIBRARY

LIBRARY



Purpose: To print all file names in the library directory.

Remarks: One file name is printed per print zone.

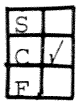
Example:

```
*LIBRARY
A1.              SHOT.SR          SQRT.SR
BACKGAMMON.SR    SUPERGUESS.SR    STOCKS.SR
CASINO.SR        SWAP.SV          SNOOP.
COMPILER.SR      FCOM.CM          BLACKJACK.SR
BANK.SR          FOOTBALL.SR      BILLBOARD.SR
KILLER.MS        K2.              MAT.SR
SNOOPY.SR        GUESS.SR         QUEEN.SR
TEST1.           HORSERACE.SR     LUNAR.SR
BATNUM.SR        HEMAN.SR         SHOT1.SR
FISCAL.SR        HELLO.SR         HELLO.SV
FISCAL.BT
*
```

WHATS

WHATS *filename*

filename: the name of a file in the programmer's directory or in the library directory.



Purpose:

To print information pertaining to *filename* at the terminal.

Remarks:

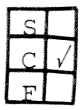
First *filename* is searched for in the programmer's directory and, if not found, is then searched for in the library directory.

Example:

```
*WHATS "ABC")
ABC  D  2039  06/14/73  09:15  (07/21/73)  00
```

filename
attributes
byte length
date created
time created
date last used
in use count

DISK



DISK

Purpose:

To obtain a count of the number of 256-word blocks still available in the programmer's directory.

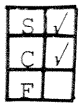
Example:

DISK)

```
USED:  332
LEFT:  193
```

This message indicates that 193 out of 525 blocks are still available for use.

DELETE



DELETE *filename*

filename: a file in the programmer's directory which is not protected (see CHATR).

Purpose: To remove a file from the programmer's directory.

Remarks:

1. This command searches the programmer's directory for the file named *filename*. If found, all references to *filename* are deleted.

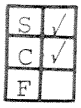
2. An error message is returned if the file cannot be found, is delete-protected, or if any attempt is made to delete files in other directories.

Example:

DELETE "TEST.SR")

The file TEST.SR is removed from the directory and the disk blocks which it formerly occupied are free for use.

RENAME



RENAME *oldfilename, newfilename*

oldfilename: a disk file in the programmer's directory.
newfilename: a new filename.

Purpose:

To search the programmer's directory for *oldfilename* and, if found, rename it to *newfilename*.

Remarks:

An error message will be printed at the programmer's terminal if:

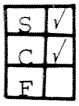
a. *oldfilename* does not exist.
b. *newfilename* already exists.
c. *oldfilename* is attribute protected.

Example:

```
*RENAME "TEST.SR", "A.SR"
```

File TEST.SR is renamed as A.SR for future referencing.

CHATR



CHATR *filename, attributes*

filename: a disk file in the programmer's directory, expressed as a string literal or string variable.

attributes: file attributes described under remarks.

Purpose:

To change, add or remove the resolution file attributes assigned to a file which already exists in the programmer's directory.

Remarks:

1. The CHATR command will not affect RDOS attributes which are not implemented under the BASIC CHATR command.
2. File attributes may be strung together in the *attributes* argument without the use of delimiting spaces or punctuation and may be expressed as a string literal or string variable.
3. The attributes listed in the BASIC CHATR command replace existing attributes, unless otherwise specified.
4. The attributes which may be used in the BASIC CHATR command are:

   P - Permanent file. The file *filename* cannot be deleted or renamed once this attribute has been assigned.

   R - Read protected. The file *filename* cannot be accessed for reading.

   W - Write protected. The file *filename* cannot be altered.

   H - Sharable. The file *filename* may be accessed by other users so long as they know the directory and file name. The file is permanent (P) and write protected (W).

   O - Sharable. The file *filename* may be accessed by other users so long as they know the directory and file name. The file is not permanent (P)

Remarks:
(Continued)

or write protected (W) and, therefore, may be deleted, written into, or renamed by other users.

E - Execute only. Other users may execute the BASIC program contained in *filename*, but are prevented from examining the program source statements. Commands such as LIST or SAVE result in an error message.

Ø - Zero. Removes current file attributes except those which are set by an RDOS CHATR command and are not included as attributes under the BASIC CHATR command. When Ø is listed with other attributes in a CHATR command, only the attributes listed are removed.

* - Asterisk. Preserve current file attributes and add those specified. The asterisk (*) may only be used in conjunction with other attributes in the argument.

Example:

```
*WHATS "TESTFILE"
TESTFILE.       D                    260
*CHATR "TESTFILE","WP"
*WHATS "TESTFILE"
TESTFILE.       WPD                  260
*
```

COMMANDS DERIVED FROM BASIC STATEMENTS

Any BASIC statement that can meaningfully be written as a keyboard command can be used in that mode. Certain statements have meaning only within the context of a program and cannot be used as keyboard commands. These commands are CHAIN, DATA, DEF, END, FOR, GOSUB, GOTO, NEXT, ON, REM, RETURN, and STOP. All other BASIC statements are implemented as keyboard commands which may be used to:

- Perform file I/O
- Perform desk calculation functions
- Dynamically debug programs

Perform File I/O

The opening and closing of files and the input/output of programs and data from files and devices can be handled by keyboard commands derived from the file I/O statements described in Chapter 7.

```
OPEN FILE (1,3), "$PTR")
READ FILE (1) , A, B, C, D, E, F, G (5))
```

Desk Calculator

The PRINT command can be used to obtain immediate results of arithmetic computations.

```
;EXP (SIN (3.4/8)))  1.51032
LET A = EXP (SIN(3.4/8))
;USING "+####.##↑↑↑↑", A)  +1510.32E-03
```

Notice that the resultant value is printed on the same line.

Desk Calculator - Using Program Values

The programmer can interrupt a running program and use the assigned values of program variables for making calculations.

```
0010 DIM A$ = (10), B$ (10)
0020 LET A$ = "IOU $10.50"
0030 B$ = "XRAY"
RUN)
(ESC)                                  ← Press ESC key
;B$(4);A$(2,3))  YOU
```

Dynamic Program Debugging

A running program can be interrupted (using ESC or by programmed STOP statements) at a number of different program points. The current values of the variables can then be checked at those points and corrections made in the program, either to statements or variables, as necessary. The programmer can then use the RUN *line no.* command to restart the interrupted program without losing either the values of the variables at the point of interruption or the newly inserted values and statements.

```
.
.
.
(ESC)                                  ← Press ESC key.
STOP AT 1100
IF A< >B THEN PRINT B,A)               ← Command conditionally provides for
.025          .5                         examination of A and B.


.
.
.
2.33333                                ← results of a series of program
5.41234                                   calculations being printed.
8.99999

(ESC)                                  ← Press ESC key.
STOP AT 0570
READ X1, X2, X3)                       ← Space over the next 3 values in
RUN 570)                                  the data block. Resume program
                                          execution at the point it was
                                          interrupted.

(ESC)                                  ← Press ESC key.
STOP AT 1100
;A)  0                                 ← Check value of variable A.

A = -1)                                ← Change the value of arithmetic
C$ = "% OF LOSS"                          variable A and string variable
RUN 505)                                  C$. Resume running at statement
                                          505.
```

Dynamic
Program
Debugging
(Continued)

```
.
.
.
20 DIM A [4,4]
.
.
.
(ESC)                          ← Press ESC key.
STOP AT 500
DIM A [3,5])                   ← Redimension
                                 array A.
```

CHAPTER 9

ADVANCED BASIC STATEMENTS AND COMMANDS

INTRODUCTION

The items described in this chapter provide the experienced programmer with the facility for:

- Error handling
- Formatted output
- Chaining
- Subroutine calls, and
- Timed input.

ON-ERR

S	√
C	
F	

ON ERR THEN *statement*

statement: any BASIC statement except FOR, NEXT, DEF, END, DATA and REM.

Purpose:

To direct the program to an error handling routine other than normal BASIC system error handling.

Remarks:

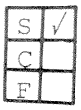
1. This statement is placed in the program prior to any statements with which the programmer's error handling routine deals. If placed at the beginning of a program, *statement* is executed for all program errors. If placed anywhere else in the program, *statement* is only executed for errors which occur after the ON ERR statement is encountered.
2. The ON ERR THEN STOP statement is used to restore system error handling and can be effectively used with ON ERR THEN *statement* to provide special error handling for selected portions of a program.
3. If *statement* is a GOSUB, then when the subroutine RETURNs, control is passed to the statement following the statement on which the error occurred. A RETRY statement should not be used in the body of the subroutine.

Example:

```
10   ON ERR THEN GOTO 1000
20   OPEN FILE (0,0), "X"
30   ON ERR THEN STOP
.
.
.
1000 OPEN FILE (0,0), "Y"
1010 GOTO 30
```

RETRY

RETRY



Purpose:

To repeat the statement which caused an error.

Remarks:

This statement can be used in conjunction with the ON ERR statement to cause program execution to return to the statement which caused the error and attempt to re-execute that statement.

Examples:

```
* 5  ON ERR THEN 100
*10  OPEN FILE (0,2), "TESTING"  ← If state-
 .                                  ment 10 causes an
 .                                  error then RETRY
 .                                  directs the program
*100 RETRY                          to repeat the state-
 .                                  ment.
 .
 .
```

Note: In this example, if statement 10 causes an error then statements 5 and 100 would cause the program to loop indefinately. The program should, therefore, include some provision for exiting from RETRY such as exiting after a certain number of failures.

DELAY

DELAY = *expr*

expr: a numeric expression which evaluates to an integer and represents time in seconds.

S	√
C	
F	

Purpose:

To delay program execution for a specified amount of time.

Remarks:

1. The DELAY statement resets the SYS(14) function to a value of zero.
2. When used in conjunction with RETRY, program execution can be postponed on an error condition before a RETRY is attempted.

Example:

```
5     ON ERR THEN 100
10    OPEN FILE (Ø,2), "THISFILE"
.
.
.
100   IF SYS (7)<>1Ø THEN 200     ← Is another user
                                    using this file?
105   I = I + 1
110   IF I>1Ø THEN GOTO 200       ← 10 RETRY attempts
                                    allowed.
120   DELAY=1                     ← One second delay
                                    before RETRY.
125   RETRY                       ←Returns to state-
                                    ment which caused
                                    error.
200   STOP
```

ON-ESC

ON ESC THEN *statement*

S	✓
C	
F	

statement: any BASIC statement except FOR, NEXT, DEF, END, DATA and REM.

Purpose:

To direct the program to a user handling routine when the ESC key is pressed.

Remarks:

1. Normally, when the ESC key is pressed any operation in progress is interrupted and the terminal is ready for input. When ON ESC THEN *statement* is executed pressing the ESC key will cause the *statement* argument from ON ESC THEN *statement* to be executed.
2. The normal handling of ESCape can be restored by the ON ESC THEN STOP statement.
3. If *statement* is a GOSUB, then when the subroutine RETURNs, control is passed to the statement following the statement on which the error occurred. A RETRY should not be used in the body of the subroutine.

Examples:

(Continued on next page)

Examples:
(Continued)

1.
```
100  ON ESC THEN PRINT X,Y,Z
  .
  .
  .
140  PRINT X
141  Y=Z
  .
  .
  .
```

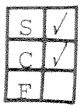
In this example, when the user presses the key during program execution, control passes to the statement on line 100 and the values of X, Y and Z are printed. After line 100 is executed, the program continues as if line 100 were not included in the program and executes the next line after the last completed before the ESC key was pressed. Therefore, if line 140 had been completed when ESC was pressed, line 100 would be executed followed by line 141.

2.
```
10   ON ESC THEN GOSUB 500
20   DIM X(2500)
21   A = 0
22   B = 0
23   C = 0
30   FOR I = 1 TO 2500
40   X(I) = A*I↑2+B*I+C
50   NEXT I
60   STOP
500  PRINT I, X(I)
510  INPUT "CONTINUE (0) OR NEW INPUTS (1)",D
520  IF D = 0 THEN RETURN
530  INPUT "NEW VALUES FOR A,B,C", A,B,C
540  RETURN
```

In this example, a RETURN from line number 520 or 540 is not to line 20 but to the line after the last executed when the ESC key was pressed.

PRINT USING

PRINT USING *format*, *expr* [,*expr*] ...



format: a string literal or string variable which specifies the format (see Remarks) for printing the items in the *expr* list.

expr: a list of one or more expressions which may include numeric variables, subscripted variables, string literals and string variables.

Purpose:

To output the values of expressions in the PRINT USING statement list using the *format* specified.

Remarks:

1. All normal PRINT formatting conventions (e.g., TAB, comma, semicolon) are ignored in a PRINT USING statement.
2. The *format* expression may have more than one format field and may include string literals as well as the following special characters which are used for formatting numeric output.

```
#
.
+
-
$
, (comma)
↑
```

a. Digit Representation (#)

For each # in the *format* field, a digit (Ø to 9) is substituted from the *expr* argument.

format	*expr*	Representation	Remarks
#####	25	ΔΔΔ25	Right justify digits in field with leading blanks.

Remarks:
(Continued)

a. Digit Representation (#) (Continued)

format	*expr*	Representation	Remarks
#####	-30	ΔΔΔ30	Signs and other non-digits are ignored.
#####	1.95	ΔΔΔΔ2	Only integers are represented; the number is rounded to an integer.
#####	598745	*****	If the number in *expr* has more digits than specified by *format*, then all asterisks are output.

b. Decimal Point (.)

The decimal character (.) places a decimal point within the string of digits in the fixed position in which it appears in *format*. Digit (#) positions which follow the decimal point are filled; no blank spaces are left in these digit positions. When *expr* contains more fractional digits than *format* allows, the fraction will be rounded to the limits of *format*. When *expr* contains less fractional digits than specified by *format*, zeroes are output to fill the positions.

format	*expr*	Representation	Remarks
#####.##	20	ΔΔΔ20.00	Fractional positions are filled with zeroes.
#####.##	29.347	ΔΔΔ29.35	Rounding occurs on fractions.

Remarks:
(Continued)

b. Decimal Point (.) (Continued)

format	*expr*	Representation	Remarks
#####.##	789012.34	********	When *expr* has too many significant digits to the left of a decimal point, a field of all asterisks, including the decimal point, is output.

c. Fixed Sign (+ or -)

A fixed sign character appears as a single plus (+) sign or minus (-) sign in either the first character position in the *format* field or in the last character position in the *format* field.

A fixed plus (+) sign prints the sign (+ or -) of *expr* in the position in which the fixed plus (+) sign is placed in *format*.

A fixed minus (-) sign prints a minus (-) sign for negative values of *expr* or a blank space for positive values of *expr* in the position in which the fixed minus (-) sign is placed in *format*.

When a fixed sign is used, any leading zeroes appearing in *expr* will be replaced by blanks, except for a single leading zero preceding a decimal point.

format	*expr*	Representation	Remarks
+##.##	20.5	+20.50	

Remarks:
(Continued)

c. Fixed Sign (+ or -) (Continued)

format	*expr*	Repre-sentation	Remarks
+##.##	1.01	+Δ1.01	Blanks precede the number.
+##.##	-1.236	-Δ1.24	
+##.##	-234.0	******	
###.##-	20.5	Δ20.50Δ	
###.##-	000.01	ΔΔ0.01Δ	One leading zero before the decimal point is printed.
###.##-	-1.236	ΔΔ1.24-	
###.##-	-234.0	234.00-	

d. Floating Sign (++ or --)

A floating sign appears as two or more plus (++) or minus (--) signs at the beginning of the *format* field. Use of the floating plus (++) sign outputs a plus or minus sign immediately before the value of *expr* with no separating blank spaces as would occur with Fixed signs. A floating minus (--) outputs either a minus or blank (for plus) immediately preceding the value.

Positions occupied in *format* by the second sign and any additional signs can be used for numeric positions in the value of *expr*.

Remarks:
(Continued)

d. Floating Sign (++ or --) (Continued)

format	*expr*	Repre-sentation	Remarks
---.##	-20	-20.00	Second and third minus signs are treated as # on output.
---.##	-200	******	Too many digits to left of decimal point.
---.##	2	ΔΔ2.00	

Note: A *format* may include a floating sign (plus or minus) or a floating $ sign (described in paragraph f.), but not both.

e. Fixed Dollar Sign ($)

A fixed $ sign appears as either the first or second character in the *format* field, causing a dollar sign ($) to appear in that position. If the dollar sign ($) is in the second position, it must be preceded by a Fixed Sign (+ or -). A fixed dollar ($) sign causes leading zeroes in the value of *expr* to be replaced by blanks.

format	*expr*	Repre-sentation	Remarks
-$###.##	30.512	Δ$Δ30.51	
$###.##+	-30.512	$Δ30.51-	

Remarks:
(Continued)

f. Floating Dollar Sign ($$)

A floating dollar sign appears as two or more dollar ($$) signs beginning at either the first or second character in the *format* field. If the dollar signs ($$) start in the second position, they must be preceded by a fixed sign (+ or -).

A floating dollar sign ($$) causes a dollar sign to be placed immediately before the first digit of the *expr* value.

Note: A *format* may include a floating dollar ($$) sign or a floating sign (plus or minus), as described in preceding paragraph d, but may not include both.

format	*expr*	Representation	Remarks
+$$$#.##	13.20	+ΔΔ$13.20	Extra $ signs may be replaced by digits as with floating + and - signs.
$$##.##	-1.0	Δ$01.00-	Leading zeroes are not surpressed in the # part of the field.

g. Separator (,)

A comma (,) separator places a comma in the fixed position in which it appears in a string of digits (#) in the *format* field.

If a comma would be output in a field of surpressed leading zeroes (blanks), then a blank space is output in the position for the comma.

Remarks:
(Continued)

g. Separator (,) (Continued)

format	*expr*	Repre-sentation	Remarks
+$#,###.##	30.6	+$ΔΔΔ30.60	Space printed for comma.
+$#,###.##	2000	+$2,000.00	
++##,###	00033	Δ+00,033	Comma is printed when leading zeroes are not surpressed.

h. Exponent Indicator (↑)

Four consecutive up-arrows (↑) are used to indicate an exponent field in *format*. The four up-arrows will be output as E+nn, where each n is a digit.

If the exponent field in *format* does not have exactly four up-arrows, then a run-time error will result.

format	*expr*	Repre-sentation	Remarks
+##.##↑↑↑↑	170.35	+17.03E+01	
+##.##↑↑↑↑	-.2	-20.00E-02	
++##.##↑↑↑↑	6002.35	+600.24E+01	

Remarks:
(Continued)

3. As previously indicated, a *format* expression may include more than one *format* field and may include string literals in addition to the special formatting characters. Values of the *expr* argument list are sequentially assigned to *format* fields.

   BASIC differentiates *format* fields from string literals by the characters that appear in *format* fields.

   For example:

"TWO FOR $1.25"	$1.25 is part of the string literal.
"TWO FOR $$$.##"	$$$.## is a *format* field in the *format* expression.
"ANSWER IS -85"	-85 are characters of the string literal.
"ANSWER IS -###"	-### is a *format* field in the *format* expression.

4. A *format* expression maybe specified by referencing a previously defined string variable; for example:

```
5   DIM S$(10)
10  LET S$="##.##"
20  PRINT USING S$, 1.5, 2
```

5. *Format* fields in a *format* expression are delimited by the use of a non-special formatting character before or after the *format* field.

Remarks:
(Continued)

field delimiter field delimiter

"#####ΔFORΔ$$###.##"

format field | string literal | format field

6. String literals may appear in the *expr* argument list of the PRINT USING statement and will be superimposed on a *format* field in the following manner :

   a. Each character of the string literal replaces a single *format* field character, which may be any of the special format characters ($,#,↑, and comma).
   b. Strings are left justified in the *format* field, and filled with spaces, if necessary.
   c. If the number of characters in the string is greater than the number of characters in the *format* field, then the string will be truncated to fit the field.

```
5 PRINT USING "###,###.##","TEST,"CHARACTER","SEVENTY-FIVE"
RUN)
TESTΔΔΔΔΔΔCHARACTERΔSEVENTY-FI
```

7. When there are more items in the *expr* argument list than *format* fields in the *format* expression then the *format* fields will be used repetitively.

   "####Δ@$###.##ΔPERΔ###"

   The first, fourth, seventh, etc., items in the *expr* argument list will be formatted using the *format* field ####.

   The second, fifth, eighth, etc., items in the *expr* argument list will be formatted using the *format* field $###.##.

Remarks:
(Continued)

The third, sixth, ninth, etc., items in the *expr* argument list will be formatted using the *format* field ###.

The embedded blank spaces, @ sign, and PER are string literals and delimit the *format* fields.

```
.
.
.
100 PRINT USING "A(#)Δ=Δ##.#",I,A(I)
.
.
.
RUN )
A(1)Δ=Δ17.9                          ← Possible output includes two format fields and two string literals.
.
.
.
100 PRINT USING "###.##Δ",I,A,B
.
.
.
RUN )
ΔΔ1.00ΔΔ17.90ΔΔ25.77Δ               ← Possible output with format expression repeated for each item in argument list.
```

PRINT FILE
USING

S	✓
C	✓
F	

PRINT FILE (file), USING *format, expr*

file: a numeric expression which evaluates to the number of a file opened in Mode 1, 2, 4 or 5 for sequential output.

format: a string literal or string variable which specifies the format (see Remarks) for outputting the items in the *expr* list.

expr: a list of one or more numeric expressions, numeric variables, string variables, and string literals whose values are written into a sequential access file.

Purpose:

To output the values of the expressions in the PRINT FILE USING statement to a previously opened file using the *format* specified.

Remarks:

The remarks for the PRINT FILE statement described in Chapter 7 and the PRINT USING statement described in this chapter are all applicable to the PRINT FILE USING statement.

CHAIN

S	✓
C	
F	

CHAIN *filename* [THEN GOTO *line no.*]

filename: a string variable or string literal evaluating to a device or a disk file.

line no.: a line number in program *filename*.

Purpose:

To run the program named in the CHAIN statement when encountered in the user's program.

Remarks:

1. When a CHAIN statement is encountered in a program, it stops execution of that program, retrieves the program named in the CHAIN statement from the specified device and file, and begins execution of the CHAINed program.
2. If the program is on disk, the system searches the programmer's directory for *filename*; if not found, the system will search the library disk directory.
3. If *filename* is found, the programmer's currently running program is cleared from memory and *filename* is loaded into memory. If *filename* is not found, the current program remains in memory.
4. The newly loaded program is run, by default, from the lowest number statement in the program unless the THEN GOTO *line no.* argument is given in the CHAIN statement to specify another line number from which execution is to begin.
5. A program must be in SAVE file format before it can be CHAINed.
6. Typically, the CHAIN statement can be used for dividing a large program into smaller programs or for running independant programs from a main program based on conditional transfer statements.

Example:
(Continued)

```
10   READ A
20   IF A > 5 THEN 60
30   IF A = 5 THEN 70
40   DATA 4,1,6,3,5
50   GOTO 10
60   CHAIN "SERVICE"
70   CHAIN "SUBR" THEN GOTO 50
```

CALL

CALL *subr* [,*expr*] ...

S	✓
C	✓
F	

subr: a positive integer representing an assembly language subroutine number.

expr: as many as eight optional arguments to be passed to the subroutine. Arguments may be arithmetic or string variables or expressions.

Purpose:

To call a subroutine written in assembly language from an Extended BASIC program.

Remarks:

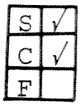
1. Dimensioned numeric variables which are used as arguments to the CALL statement must include subscripts.
2. Details for creating assembly language subroutines which may be CALLed from Extended BASIC programs are provided in Appendix B.

Example:

```
5    LET A = 12
10   LET B = A * 2
15   CALL 33,A,B          ← CALL subroutine
.                            33 and use the
.                            values of A and
.                            B as arguments
                             for the subroutine.
```

TIME

TIME = *expr*



expr: a numeric expression which evaluates to an integer and represents time in seconds.

Purpose:

To establish the time limit for timed input (TINPUT) operation.

Remarks:

1. Assigning a value to TIME sets the SYS(14) function to the value of *expr*.
2. The value of SYS(14) is decremented at the RDOS clock tick rate (1/10 of a second per tick) from the time a TINPUT statement is executed.
3. Decrementing of SYS(14) stops when the programmer responds to the TINPUT prompt. Decrementing of SYS(14) is resumed when the next TINPUT is executed.
4. If the programmer does not respond to the TINPUT prompt before the SYS(14) function has decremented to zero, then an error message is printed at the terminal and the program stops, unless an ON ERR THEN statement is used.
5. TIME may be reset to another value and may appear as often as required by the program logic.

Example:

(Continued on next page)

Example:
(Continued)

```
*LIST
0010 DIM A$[50]
0020 PRINT "LET'S TEST YOUR RECALL SPEED"
0030 PRINT
0040 TIME =10
0050 TINPUT "WHAT COLOR IS YOUR MOTHER'S EYES? ",A$
0060 GOSUB 0140
0070 TINPUT "WHAT'S YOUR SOCIAL SECURITY NO.? ",A$
0080 GOSUB 0140
0090 TINPUT "HOW OLD IS YOUR FATHER? ",A$
0100 GOSUB 0140
0130 GOTO 0190
0140 LET I=I+1
0150 LET A[I]=(10-SYS(14))
0160 PRINT "TIME USED- ";A[I];" SECONDS"
0170 TIME =10
0180 RETURN
0190 FOR J=1 TO I
0200   LET B=B+A[J]
0210 NEXT J
0220 LET C=B/I
0230 PRINT "AVERAGE RESPONSE TIME= ";C;" SECONDS"

*RUN
LET'S TEST YOUR RECALL SPEED

WHAT COLOR IS YOUR MOTHER'S EYES? BROWN
TIME USED-  6  SECONDS
WHAT'S YOUR SOCIAL SECURITY NO.? 118234567
TIME USED-  10  SECONDS
HOW OLD IS YOUR FATHER? 63
TIME USED-  4  SECONDS
AVERAGE RESPONSE TIME=  6.66667  SECONDS

END AT 0230
*
```

TINPUT

TINPUT["*string lit*",]{*var* / *svar*} [{,*var* / ,*svar*}] ... [;]

var and *svar*: a list of variables separated by commas or carriage returns.

"string lit": a message or prompt.

S	✓
C	✓
F	

Purpose:

To assign the values supplied by input from the terminal to a list of variables, within a prescribed time.

Remarks:

1. INPUT statement remarks (Chapter 3) are applicable to TINPUT.
2. The TINPUT statement is used in conjunction with the TIME= statement and the SYS(14) function.
3. The TIME= statement sets SYS(14) to the value, in seconds, allowed for the programmers respond to the TINPUT prompt.
4. The value of SYS(14) is decremented at the RDOS clock tick rate (1/10 of second per tick) from the time a TINPUT statement is executed.
5. If the programmer does not respond to the TINPUT prompt, before the SYS(14) function has decremented to zero, then an error message is printed at the programmer's terminal and the program stops, unless an ON ERR THEN statement was previously executed.
6. Decrementing of SYS(14) stops when the programmer responds to the TINPUT prompt.
7. A DELAY statement upon completion clears the value of the SYS(14) function to zero.

Example:

See TIME for an example of TINPUT usage.

APPENDIX A

ERROR MESSAGES

Extended BASIC error messages are printed as two digit codes, followed by a brief explanatory message. There are three categories of errors which may occur when operating Extended BASIC under RDOS.

1. Errors recognized by BASIC during program input.

   If an error is detected in a statement input from a terminal, the error message refers to the last statement typed.

   If the statement in error was input from a file or other input device, BASIC prints the incorrect statement followed by the error message.

   All syntax errors are recognized during program input.

   The form of the error message is:

   ERROR *xx text*

   xx: a two-digit decimal error code.
   text: a brief description of the error.

2. Run-time errors (except file I/O).

   BASIC system run-time errors cause printout of an error message in the following form:

   ERROR *xx* AT *yyyy text*

   xx: a two-digit decimal error code.
   yyyy: the line number at which the error was detected.
   text: a brief description of the error.

3. File I/O errors

Error messages related to file I/O are formatted as follows:

I/O ERROR *xx* (AT *yyyy*) *text*

xx: a two-digit decimal error code.
yyyy: the line number at which the file I/O error was detected.
text: a brief description of the error.

The following table itemizes the Extended BASIC error codes and their explanations.

BASIC Error Messages

Code	Text	Meaning	Example
00	FORMAT	unrecognizable statement format	A==2
01	CHARACTER	illegal ASCII character or unexpected character	RUN $100 ENTER #$LPT"
02	SYNTAX	invalid argument type	20 IF SIN(A$)=0...
03	READ/DATA TYPES	READ specifies different data type than DATA statement	10 DIM A$(10) 20 READ A$ 30 DATA 12 RUN
04	SYSTEM	hardware or software malfunction	
05	STATEMENT NUMBER	statement number not in the range: $1 \leq n \leq 9999$	0010 GOTO 81373
06	EXCESSIVE VARIABLES	attempt to declare more than 286 variables	
07	COMMAND I/O	attempt to execute a command from a file (and not in BATCH mode)	ENTER "ABC" and file ABC contains a LIST command
08	SINGULAR MATRIX	attempt to access via MAT a one dimension list.	10 DIM A(10) 20 MAT INPUT A
09	(NOT USED)		

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
10	RESERVED FILE IN USE	another user has control of the specified I/O device	User A: ENTER "$PTR" User B: ENTER "$PTR"
11	PARENTHESES	parentheses in an expression are not paired	A = ((B-C)
12	COMMAND	keyword unrecognizable	10 LETT A = 10
13	LINE NUMBER	attempt to delete or list an unknown line; attempt to transfer to an unknown line	100) 10 GOTO 100 RUN
14	PGM OVERFLOW	not enough storage to ENTER source program	ENTER "ABC"
15	END OF DATA	not enough DATA arguments to satisfy READ	10 READ A,B,C 20 DATA 91,21 RUN
16	ARITHMETIC	value too large or too small to evaluate or a divide by 0	A = 1234E + 66 ;A ↑ 20 ;1/0
17	(NOT USED)		
18	GOSUB NESTING	more nested GOSUB's than specified at SYSGEN	
19	RETURN - NO GOSUB	RETURN statement encountered without a corresponding GOSUB	10 RETURN RUN

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
20	FOR NESTING	more nested FOR's than specified at SYSGEN	
21	FOR - NO NEXT	unexecutable FOR-NEXT loop; FOR without a NEXT	FOR I = 1 TO 0 STEP 1
22	NEXT - NO FOR	NEXT statement encountered without a corresponding FOR	10 NEXT I RUN
23	DATA OVERFLOW	not enough storage left to assign space for variables	10 DIM A(300000) RUN
24	NO AVAILABLE CHANNELS	channel limit specified at SYSGEN time has been reached	10 OPENFILE(0,3),"T"
25	OPTION	feature specified not available (SYSGEN)	MAT PRINT A
26	PGM/DATA OVERFLOW	attempt to LOAD or RUN a SAVEd file which is too large for available storage	LOAD "ABC"
27	FILE NUMBER	invalid file designation in an I/O statement	OPEN FILE (9,0),"TEST"
28	DIM OVERFLOW	an array or string exceeds its initial dimensions	10 DIM A(2,2) 20 MAT A = ZER(5,5)
29	EXPRESSION	an expression is too complex for evaluation	A = (((A+1) + ((A-7+3) * 3) + RND (0))

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
30	MODE	invalid mode designation in an I/O statement	OPEN FILE (0,12),"TEST"
31	SUBSCRIPT	subscript exceeds array's dimension	10 DIM A(2) ;A (1,30) RUN
32	UNDEFINED FUNCTION		10 A = FNA(B)
33	FUNCTION NESTING	the nesting of too many defined functions	
34	FUNCTION ARGUMENT	argument range exceeded	A = 1234 ;A↑34652 PAGE = 200 DELAY = -1 TIME = -1000
35	ILLEGAL MASK	PRINT USING statement is illegal	;USING "A",A
36	STRING SIZE	the size of the string exceeds PAGE specification	PAGE = 10 ;"AAAAAAAAAAA"
37	USER ROUTINE	CALL statement specifies a user routine not in storage	10 CALL 2 RUN
38	(NOT USED)		

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
39	DUP MATRIX	same matrix appears on both sides of a MAT multiply or transpose statement.	10 DIM A(10,10) 20 MAT A = A * A
40	MATRICES SIZES	matrices have different sizes	10 DIM A(10,10) 20 DIM B(20,20) 30 MAT A = B RUN
41	UNDIMEN-SIONED VARIABLE	attempt to use an undimensioned matrix	A = 0 MAT PRINT A
42	FILE ALREADY OPEN	two OPEN statements without an intervening CLOSE	OPEN FILE (0,2),"$LPT" OPEN FILE (0,2),"$LPT"
43	MATRIX NOT SQUARE	attempt to invert a non-square matrix	10 DIM A(20,30) 20 MAT B = INV(A) RUN
44	FILE NOT OPEN	an attempt to read/write a file which has never been opened	DIM A$(10) WRITE FILE (0),A$ INPUT FILE(0),A
45	DATA > LRECL	logical record length limit exceeded	DIM A$(300) OPEN FILE(0,1)"ABC" WRITE FILE(0),A$
46	INPUT	too many responses to [MAT] INPUT	INPUT A ? 1,2,3
47	MODE	input file opened for writing or output file opened for reading	OPEN FILE(0,1),"TEST" READ FILE(0),A

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
48	NOT A CORE IMAGE FILE	filename specified in LOAD CHAIN, or RUN *filename* command not created by SAVE.	LOAD "TEST.SR"
49	NO ROOM FOR DIRECTORY	FILES or LIBRARY commands cannot find 256 words in user program storage to read disk directory	10 DIM A(8000) RUN FILES
50	INVALID OPERATOR COMMAND	Attempt to execute a privileged command	(see System Manager's Guide)
51	USER NOT ON SYSTEM	attempt to send message to an inactive or non-existant user.	MSGΔ#$$#ΔHELLO
52	USER IN NOMSG STATE	attempt to send message to user whose terminal is in NOMSG state	
53	RENUMBER	an incorrect line number is encountered during execution of a RENUMBER command	10 GOTO 100 RENUMBER
54	STATEMENT LENGTH	more than 132 characters in either internal or ASCII format due to expansion	10 ON A GOTO 1,1,1,.. LIST 20 5"....."
55	EXECUTE-ONLY	attempt to examine a program originating from a file with the execute-only attribute	ENTER "FRED:TEST" LIST

BASIC Error Messages (Continued)

Code	Text	Meaning	Examples
56	RANGE	attempt to reference a random record beyond 262144	5 N = 300000 10 OPEN FILE(0,0),"T" 20 READ FILE(0,N),A
57	(NOT USED)		
58	INCOMPATIBLE CORE IMAGE FILE	attempt to LOAD a core image file SAVEd under a different floating point precision	LOAD "TEST.CI"
59	ZERO STEP	FOR-NEXT with STEP 0	10 FOR *=0TO50 STEP I 20 NEXT I RUN
60	TIME-OUT	timed input decremented to zero	10 TIME = 30 20 INPUT A
61	INVALID DECIMAL STRING	attempt to perform string arithmetic with non-numeric characters	10 DIM A$(80),B$(80), C$(80) 20 INPUT A$,B$ 30 C$ = A$ * B$ RUN ?ABC?5430
62	PRECISION OVERFLOW	the result of string arithmetic requires more than 18 digits for precision representation	
63	MAX SHARED DIRECTORIES	number of sharable directories in use exceeds the number specified at SYSGEN	ENTER "FRED:A"
64		(see System Manager's Guide)	

File I/O Error Messages

Code	Text	Meaning
01	ILLEGAL FILE NAME	A to Z, 0 to 9 and $
03	ILLEGAL COMMAND FOR DEVICE	INIT "$PTR", WRITE to $CDR
06	END OF FILE	Attempt to read beyond EOF marker.
07	READ PROTECTED FILE	Attempt to read from a read protected file.
08	WRITE PROTECTED FILE	Attempt to write to a write protected file.
09	FILE ALREADY EXISTS	Attempt to create an existent file.
10	FILE NOT FOUND	Attempt to reference a non-existent file.
11	PERMANENT FILE	Attempt to alter a permanent file.
12	ATTRIBUTE PROTECTED	Illegal attempt to change file attributes.
13	FILE NOT OPENED	Attempt to reference an unopened file.
17	UFT IN USE	System error.
18	LINE LIMIT	Line limit exceeded on read or write line.
20	PARITY	Parity error on read line.
23	NO FILE SPACE	Out of disk space. Delete files to make more room.
24	READ ERROR	File read error.
25	SELECT STATUS	Unit not ready or is write protected.
29	DIFFERENT DIRECTORIES	Files specified on different directories.
30	ILLEGAL DEVICE CODE	Device not in system or illegal device code.

File I/O Error Messages (Continued)

Code	Text	Meaning
38	INSUFFICIENT CONTIGUOUS BLOCKS	Insufficient number of free contiguous disk blocks. Reorganize partition.
41	NO MORE DCB'S	Attempt to open more devices or directories than are configured in the operating system.
42	ILLEGAL DIR SPECIFIER	Illegal directory specifier.
43	UNKNOWN DIR SPECIFIER	Directory specifier unknown.
44	DIR TOO SMALL	Directory is too small (Operator only).
45	DIR DEPTH	Directory depth exceeded (Operator only).
46	DIR IN USE	Released directory in use by other program.
47	LINK DEPTH	Link depth exceeded.
48	FILE IN USE	Contact System Operator if file is in your directory.
52	FILE POSITION	
54	DIR NOT INITIALIZED	Directory/device not initialized.

APPENDIX B

CALLING AN ASSEMBLY LANGUAGE SUBROUTINE FROM EXTENDED BASIC

It is possible to call a subroutine written in assembly language from an Extended BASIC program. The format of the BASIC call is:

```
CALL sub#[, A1, ..., An]
```

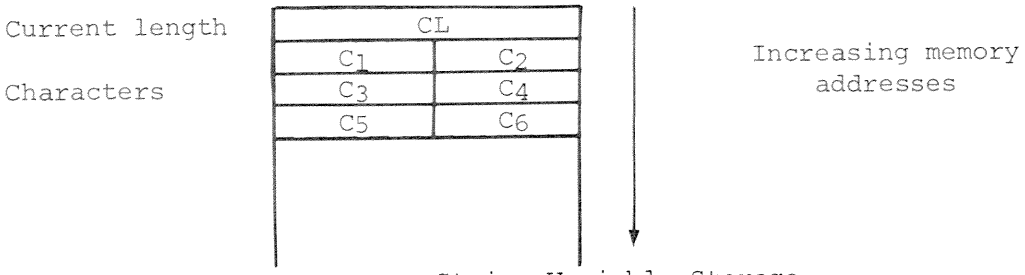
where: sub# is a numeric expression evaluating to a positive integer (in the range 0 to 32767) representing the subroutine number.

$A_1,\ldots,A_n$ are optional arguments to be passed to the subroutine (n must be in the range 1 to 8) and may be arithmetic variables or expressions, or string variables or expressions. Dimensioned numeric variable names should not appear alone, i.e., without subscripts. (Statement numbers are not permitted as arguments.)

Character String Storage and Definitions

The assembly language programmer should be aware of the following information if he wishes to handle character strings in a CALLed subroutine. BASIC keeps a count of the number of characters currently defined in each string variable (referred to as the current length of the string variable). A current length is stored as part of a header immediately preceding the contents of each string variable. (See illustration on next page) The current length must be updated each time characters are added to or taken away from the string variable.

Character String Storage and Definitions (Continued)



String Variable Storage

In the following examples, assume that A$ is dimensioned to 10, and A$ = "ABCDE". The current length of A$ is 5.

A substring is defined as any contiguous part of a string variable. For example:

A$(2,4) and A$ are substrings of A$

The current length of a substring is defined as the number of defined characters within the substring. For example, the current length of A$(4,7) is 2, if only A$(4,4) and A$(5,5) are defined.

The maximum length of a substring is defined as the number of character positions within the substring. For example, the maximum length of substring A$(4,7) is 4.

Linking the Assembly Language Subroutine

Assembly language subroutines must be submitted to the System Manager at system load time. The subroutines are input to the relocatable loader when the BASIC system save file is created. The user must include a subroutine table with his subroutines. The table must have the entry point SBRTB. Improper use of assembly language subroutines, system calls, or task calls can crash the system.

The subroutine table is a list of all assembly language subroutines available to a BASIC program. For each assembly language subroutine a four-word list is required in the table containing the following:

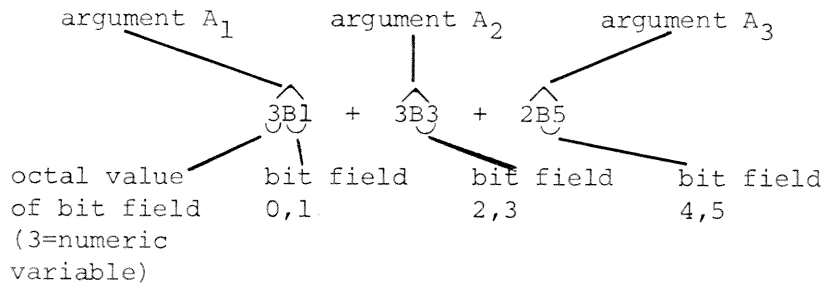
subroutine number
subroutine entry point
number of arguments
argument control word

The table is terminated by using a subroutine number of -1.

The argument control word is used by BASIC to give run-time error checking on the types of arguments. The argument control word is divided into eight two-bit fields for the eight possible arguments $A_1 \ldots A_8$. The value of the two bit field determines the allowable argument.

00_2 ← argument may be any string expression
01_2 ← argument must be a string variable
10_2 ← argument may be any numeric expression
11_2 ← argument must be a numeric variable

The argument control word is written in an assembly language program such that the arguments are connected by a plus (+) sign and are described as shown in the following example.



BASIC calls the assembly language subroutines by the sequence:

```
        LDA     2,.+2       ; AC2 POINTS TO TOP OF ADDRESS LIST
        JMP     <SUB>       ; JMP TO ASSEMBLY LANGUAGE SUBROUTINE
        ADLST

ADLST:  <arg A1>
        <arg A2>
           .
           .
           .
        <arg An>
        JMP     BASIC       ; RETURN TO BASIC INTERPRETER
```

If A_n is a substring of a string variable, the address list contains the address of the string descriptor words, which contain the following information:

word 1:	byte address of the first character of the substring
word 2:	current length of the substring
word 3:	maximum length of the substring
word 4:	word address of the current length of the string variable

If A_n is a string expression, the address list contains the address of the string descriptor words, which contain the following information:

word 1:	byte address of the first character of the string
word 2:	length of the string

If A_n is a numeric variable, the address list contains the storage address of the variable. (All numeric variables are represented in standard floating point format.)

If A_n is a numeric expression, the address list contains the storage address of the value of the expression.

The following is an example of a subroutine, and its subroutine table. The argument list in a BASIC call to this subroutine must match the argument control word specified in the subroutine table.

```
CALL  1,B,C          ← legal
CALL  1,A,B          ← legal

CALL  1,B*2,C        ← not legal (arg A1 must be a numeric
                       variable)
CALL  2,A,B          ← not legal (there is no subroutine
                       no.2)
```

```
                    .TITLE  SBRTB     ; BASIC ASSEMBLY LANGUAGE SUBROUTINES
                    .ENT    SBRTB     ; ENTRY POINT :  SBRTB
                    .NREL             ; NORMAL RELOCATABLE CODE

; SUBROUTINE TABLE

SBRTB:      1                         ; SUBROUTINE #
            SUB1                      ; SUBROUTINE ENTRY POINT
            2                         ; NUMBER OF ARGUMENTS
            3B1 + 3B3                 ; ARGUMENT CONTROL WORD, BOTH ARGS ARE
                                      ; NUMERIC VARIABLES
            -1                        ; END OF TABLE


SUB1:
; CALLING SEQUENCE:   CALL 1,A,B
; THIS ROUTINE IS THE EQUIVALENT OF LET B = A.
; THIS ROUTINE IS NOT REENTRANT
            STA     2,RET             ; SAVE ADDRESS LIST
            LDA     3,0,2             ; ADDRESS OF ARG 1
            LDA     3,0,3             ; WORD 1 OF ARG 1
            LDA     2,1,2             ; ADDRESS OF ARG 2
            STA     3,0,2             ; WORD 1 OF ARG 1 TO WORD 1 OF ARG 2
            LDA     2,RET             ; ADDRESS LIST
            LDA     3,0,2             ; ADDRESS OF ARG 1
            LDA     3,1,3             ; WORD 2 OF ARG 1
            LDA     2,1,2             ; ADDRESS OF ARG 2
            STA     3,1,2             ; WORD 2 OF ARG 1 TO WORD 2 OF ARG 2
            LDA     3,RET             ; ADDRESS LIST = RETURN ADDRESS -2
            JMP     2,3               ; RETURN TO BASIC (2 = NO. OF ARGS)
RET:        .BLK    1
            .END
```

An illegal CALL, causing error 17, will result from an attempt to pass a variable in the CALL that does not have a previously assigned value. All variables passed in the CALL must have been previously assigned values even if their current value is not to be used in the CALLed subroutine.

Several subroutines are available in BASIC to help the user in manipulating numbers and character strings. The pointers to the routines are in page zero and should be declared as displacement externals.

Linking the Assembly Language Subroutine (Continued)

Routines	RESULT*
.FIX	Converts floating point number in AC0-AC1 to an integer in AC0-AC1. If there is overflow, the largest possible integer is returned in AC0-AC1. Bit 0 of AC0 is the sign of the number. Bit 0 of AC1 is a significant bit. There are two returns from .FIX return 1: overflow return 2: OK
.FLOT	Converts an integer in AC0-AC1 to floating point format in AC0-AC1.
.ADDF F0+F1 .SUBF F0-F1 .MPYF F0*F1 .DIVF F0/F1	Arithmetic routines to perform floating point add, subtract, multiply, divide. In each routine, AC0-AC1 initially contains the floating point value of F1 and AC2 contains the address of the value of F0. The result is returned in AC0-AC1. Underflow returns a zero result; overflow results in error number 16.
.MPY A1*A2→A0,A1 .MPYA A0+A1*A2→A0,A1	In the integer multiply routines, AC1 contains the unsigned integer multiplicand and AC2 contains the unsigned integer multiplier. The result is a double length product with high-order bits in AC0 and low-order bits in AC1. Contents of AC2 are unchanged. The difference between the routines is that .MPYA adds the result of the multiplication to the contents of AC0.
.DVD (A0,A1)/A2→A1,A0 .DVDI A1/A2→A1,A0	In the integer divide routines the dividend is in AC1 (single-length) or in AC0 and AC1 (double-length with high order bits in AC0). The divisor is in AC2 and the result is left with the quotient in AC1 and the remainder in AC0. Contents of AC2 are unchanged.

*In systems having floating point hardware, the floating point number is stored and returned in the Floating Point Accumulator (FPAC) rather than in AC0-AC1.

Routine	Result
.MOST	Moves the character string described by the string descriptor words in AC0, AC1 to the substring described by the string descriptor words in the page zero memory locations labeled TR3, TR4, TR5, TR6. Before a JSR to MOST, these accumulators and memory locations should be loaded as follows: AC0 - byte address of the first character of the source string AC1 - length of the source string TR3 - byte address of the first character of the destination string TR4 - current length of the destination substring TR5 - maximum length of the destination substring TR6 - word address of the current length of the destination string variable.

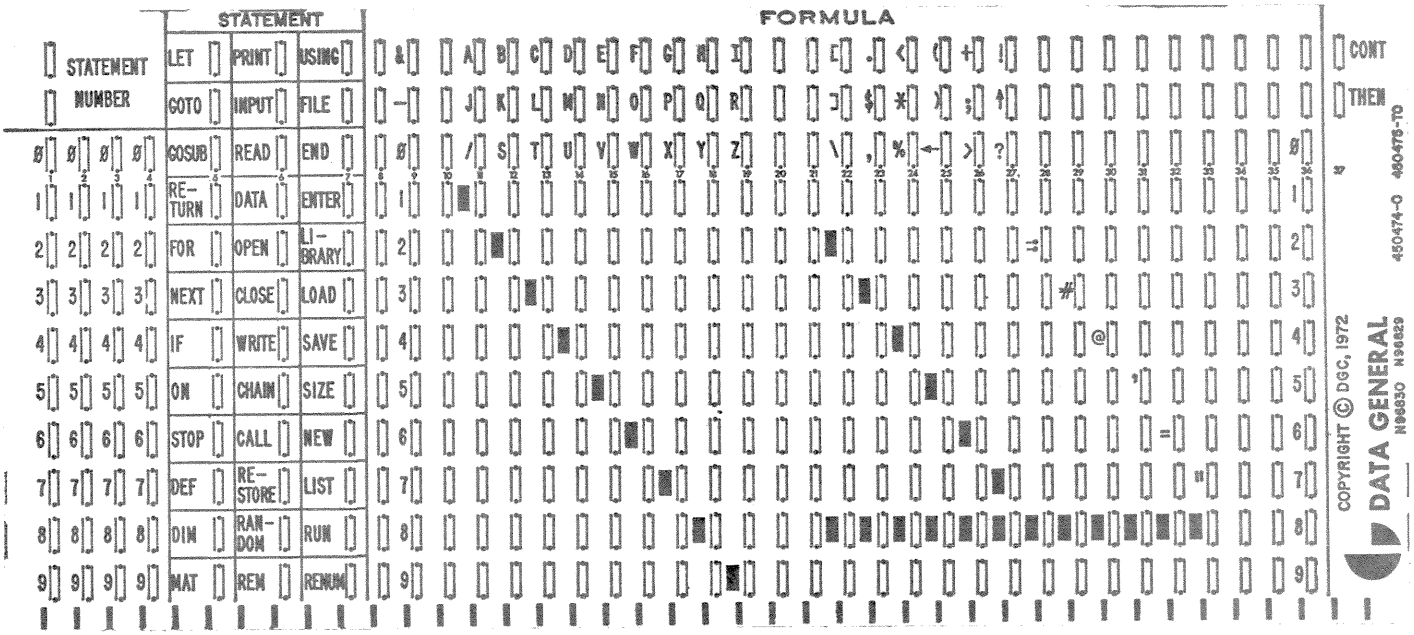
TR3, TR4, TR5 and TR6 should be declared as displacement externals in the assembly language subroutine. MOST automatically updates the current length of the destination string variable. Subroutine MOST has two returns. Return at CALL + 1 means the character string move was terminated by the source string becoming empty.

Return at CALL + 2 means the move was terminated by the destination substring becoming full.

APPENDIX C

PROGRAMMING ON MARK-SENSE CARDS

Source programs may be written on Data General's Extended BASIC mark-sense programming cards for input to the mark sense card reader. The Data General Extended BASIC mark-sense programming card is a 37-column card as shown below.

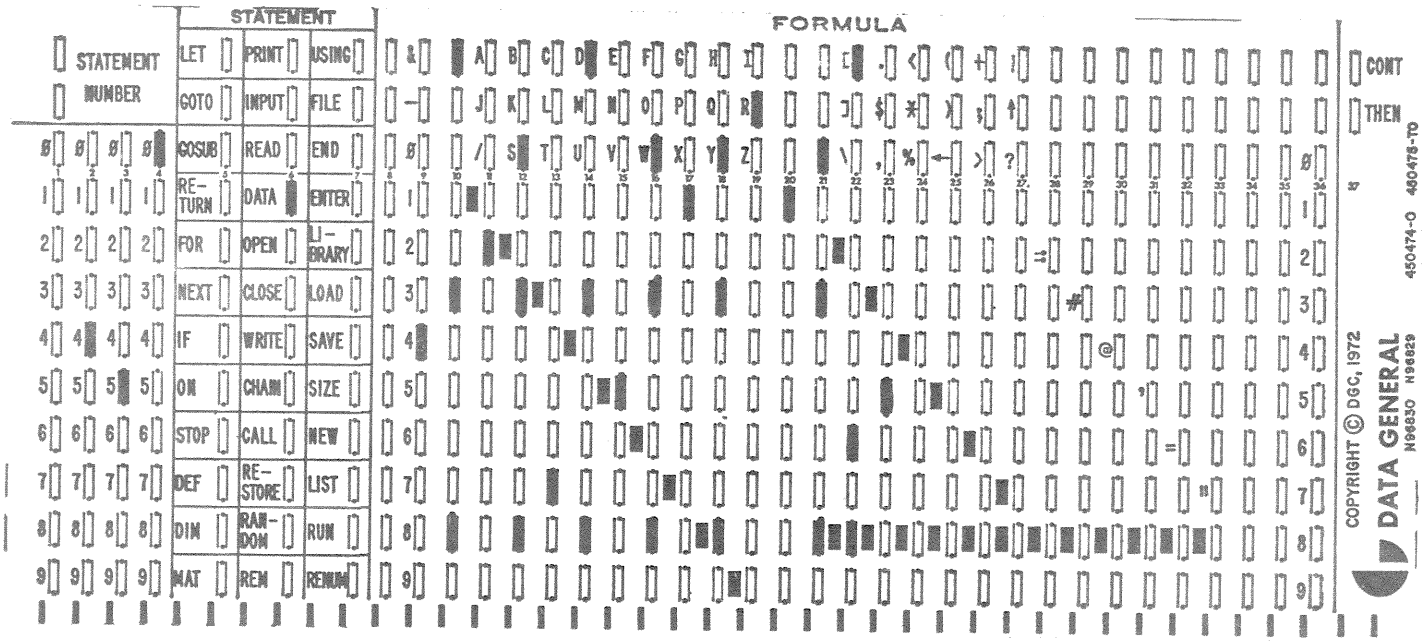


A stack of mark-sense cards may be read as a file and requires an EOF card at the end, which is a card with a single column in which all rows are marked. When in Batch mode, entire jobs may be entered from the card reader. Such card decks must conform to the Batch job control formats. Keyboard commands are suitable card input in Batch mode only. Consult with the System Manager for detailed Batch processing information.

The mark-sense reader has an option that permits either markings or punches to be read. Users having the option may punch mark-sense cards. Marked and punched cards may be intermixed in a deck and a single card may contain both markings and punches.

The BASIC statement field of the mark-sense card, as shown above, is three columns which allows all possible combinations of statement keywords. For example, MAT appears in the first column of the field, WRITE in the second, and FILE in the third, permitting the user to indicate a MAT WRITE FILE statement.

A single Extended BASIC statement or part of a statement may be written on a single card. Cards are marked with No. 2 pencil in the appropriate column; for example, the statement 450 DATA 4.2, 3.5, 1, -1, +5 would appear as:



Part of a statement may be written on one card and continued on the next by marking the CONT box in the upper righthand corner of the first card and continuing the statement on the card following beginning with the FORMULA section.

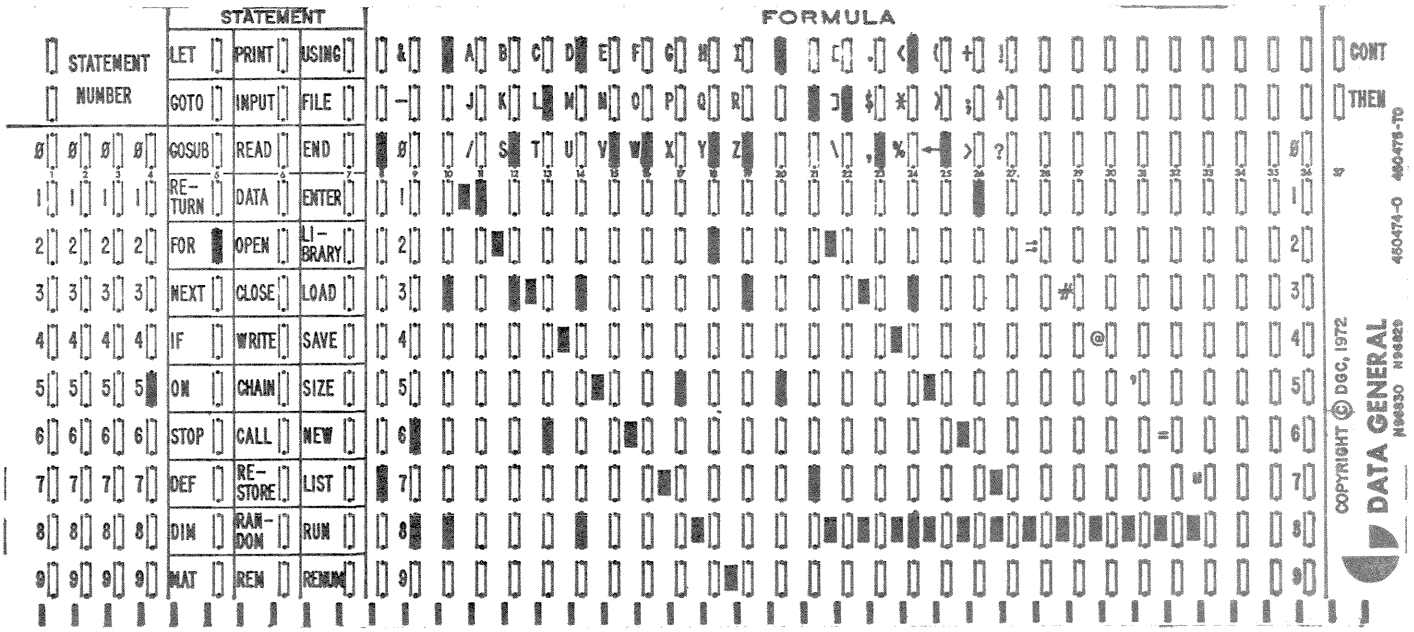
When writing an IF or an ON statement, the programmer writes one card containing the IF or ON expression and marks the THEN box in the upper righthand corner of card. The programmer continues the THEN clause on the next card beginning in the FORMULA section.

The FORMULA section of each card must be filled out in Hollerith code (see Appendix D), and programmers familiar with punched cards will have no difficulty with the format. To assist any programmers who are not familiar with punched cards, each card contains a key indicating the lines that must be marked for each character. For each character, a box on the horizontal

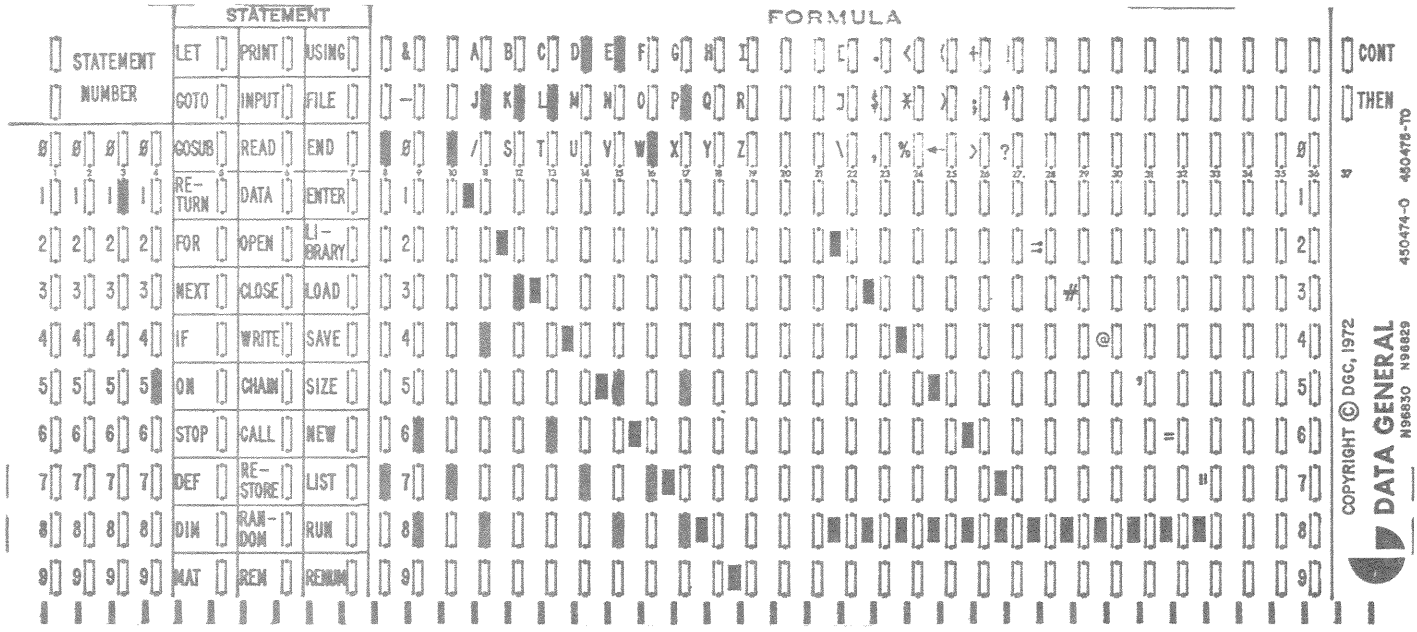
line on which the character appears must be marked. In addition, in the horizontal lines immediately below the character, a line must be marked if a square appears on the lefthand side. If we use punched card notation, the top line is designated 12, the second from the top is designated 11, and the other lines are numbered from 0 through 9. To indicate 4, put a mark on line 4; to indicate *, put marks on lines 11-4-8; to indicate #, put marks on lines 3-8, etc.

To further illustrate the use of mark-sense cards, the following source code is shown on cards:

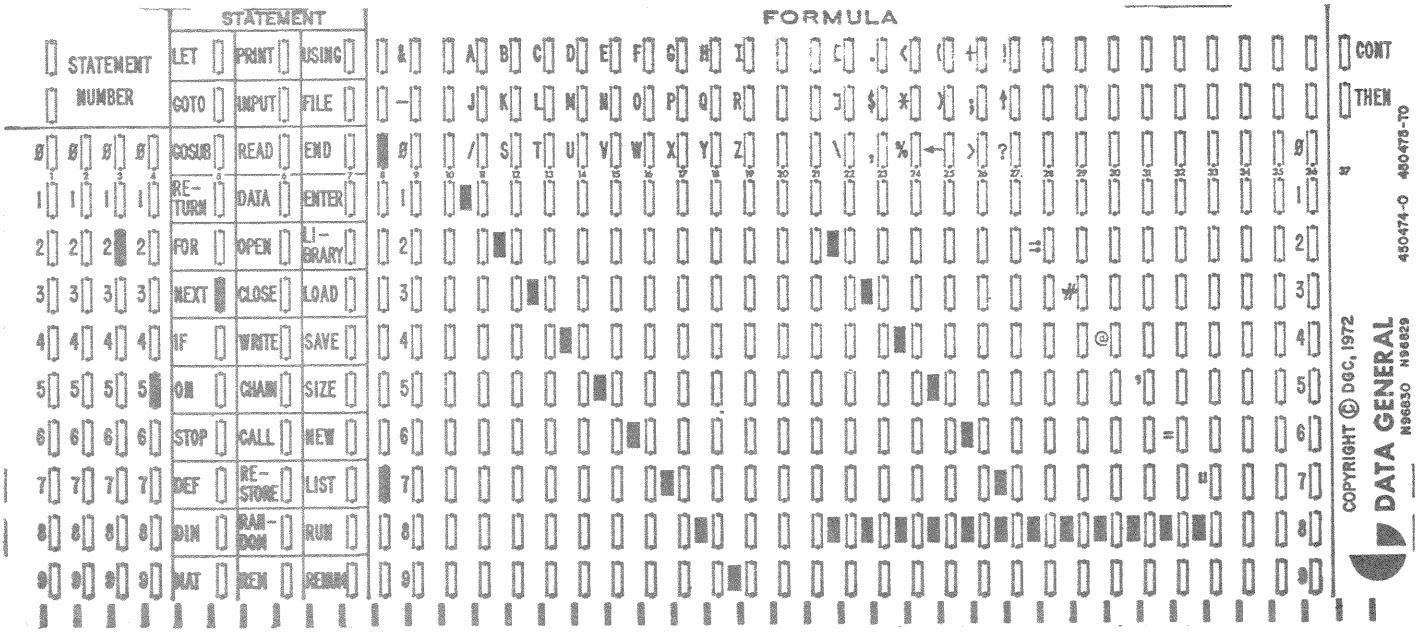
```
5     FOR X = .1 TO .005 STEP -0.01
15    X = X * LOG(X)
25    NEXT X
```



5FORX=.1TO.005STEP-0.01



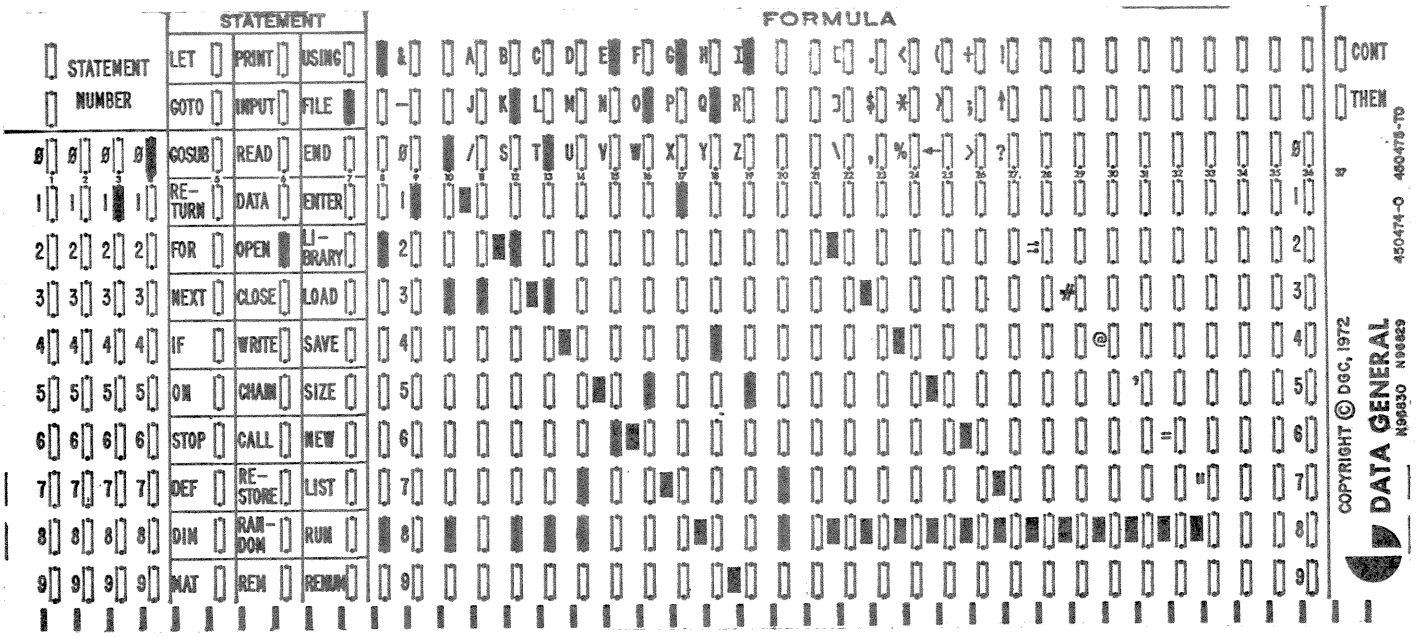

```
15 X=X*LOG(X)
```



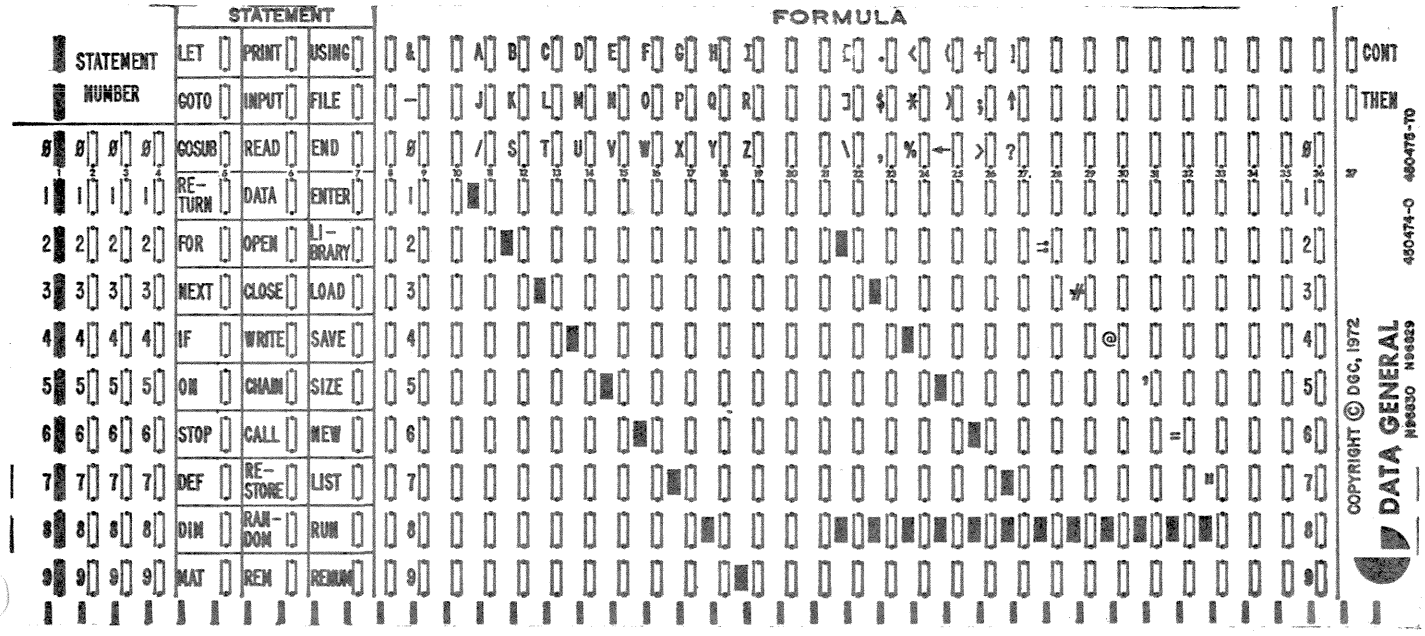

```
25  NEXTX
```

Following is a card showing the statement:

10 OPEN FILE [1,3], "FNAME"



Following is an end-of-file card with all rows of column 1 marked.



APPENDIX D

HOLLERITH CHARACTER SET

Character	Lines		
0	0	-	-
1	1	-	-
2	2	-	-
3	3	-	-
4	4	-	-
5	5	-	-
6	6	-	-
7	7	-	-
8	8	-	-
9	9	-	-
A	12	1	-
B	12	2	-
C	12	3	-
D	12	4	-
E	12	5	-
F	12	6	-
G	12	7	-
H	12	8	-
I	12	9	-

Character	Lines		
J	11	1	-
K	11	2	-
L	11	3	-
M	11	4	-
N	11	5	-
O	11	6	-
P	11	7	-
Q	11	8	-
R	11	9	-
S	0	2	-
T	0	3	-
U	0	4	-
V	0	5	-
W	0	6	-
X	0	7	-
Y	0	8	-
Z	0	9	-
[	12	2	8
.	12	3	8

HOLLERITH CHARACTER SET (Continued)

Character	Lines		
<	12	4	8
(	12	5	8
+	12	6	8
!	12	7	8
]	11	2	8
$	11	3	8
*	11	4	8
)	11	5	8
;	11	6	8
↑	11	7	8
\	0	2	8
, (comma)	0	3	8
%	0	4	8
←	0	5	8
>	0	6	8
?	0	7	8
:	2	8	-
#	3	8	-
@	4	8	-

Character	Lines		
'	5	8	-
=	6	8	-
"	7	8	-
&	12	-	-
- (minus)	11	-	-

APPENDIX E

ASCII CHARACTER SET

CHARACTER	OCTAL	DECIMAL
NULL	000	0
↑A	001	1
↑B	002	2
↑C	003	3
↑D	004	4
↑E	005	5
↑F	006	6
↑G	007	7
↑H	010	8
TAB(↑I)	011	9
LINE FEED(↑J)	012	10
VERT. TAB(↑K)	013	11
FORM FEED(↑L)	014	12
CARRIAGE RETURN(↑M)	015	13
↑N	016	14
↑O	017	15
↑P	020	16
↑Q	021	17

CHARACTER	OCTAL	DECIMAL
↑R	022	18
↑S	023	19
↑T	024	20
↑U	025	21
↑V	026	22
↑W	027	23
↑X	030	24
↑Y	031	25
↑Z	032	26
ESC or ALT MODE or CTRL-SHIFT-K	033	27
CTRL-SHIFT-L	034	28
CTRL-SHIFT-M	035	29
CTRL-SHIFT-N	036	30
CTRL-SHIFT-O	037	31
SPACE	040	32
!	041	33
"	042	34
#	043	35

ASCII CHARACTER SET (Continued)

CHARACTER	OCTAL	DECIMAL
$	044	36
%	045	37
&	046	38
'(apostrophe)	047	39
(	050	40
)	051	41
*	052	42
+	053	43
, (comma)	054	44
- (minus)	055	45
.	056	46
/	057	47
0	060	48
1	061	49
2	062	50
3	063	51
4	064	52
5	065	53
6	066	54
7	067	55
8	070	56
9	071	57
:	072	58
;	073	59

CHARACTER	OCTAL	DECIMAL
<	074	60
=	075	61
>	076	62
?	077	63
@	100	64
A	101	65
B	102	66
C	103	67
D	104	68
E	105	69
F	106	70
G	107	71
H	110	72
I	111	73
J	112	74
K	113	75
L	114	76
M	115	77
N	116	78
O	117	79
P	120	80
Q	121	81
R	122	82
S	123	83

ASCII CHARACTER SET (Continued)

CHARACTER	OCTAL	DECIMAL
T	124	84
U	125	85
V	126	86
W	127	87
X	130	88
Y	131	89
Z	132	90
[	133	91
\ (SHIFT-L)	134	92
]	135	93
↑	136	94
← or _	137	95
'	140	96
a	141	97
b	142	98
c	143	99
d	144	100
e	145	101
f	146	102
g	147	103
h	150	104
i	151	105
j	152	106

CHARACTER	OCTAL	DECIMAL
k	153	107
l	154	108
m	155	109
n	156	110
o	157	111
p	160	112
q	161	113
r	162	114
s	163	115
t	164	116
u	165	117
v	166	118
w	167	119
x	170	120
y	171	121
z	172	122
{	173	123
|	174	124
}	175	125
~ (tilde)	176	126
RUBOUT or DELETE	177	127

APPENDIX F

STATEMENT, COMMAND AND FUNCTION SUMMARY

F.1 COMMONLY USED BASIC STATEMENTS

Formats and Descriptions	S	C	F	Page Ref.
DATA {val / "string lit"} [{,val / ,"string lit"}] ... Defines data to be used by READ and MAT READ.	✓			3-8
DEF Used with FNa(d) function to define a user function.	✓			4-17
DIM {svar(m) / array(m) / array(row,col)} [{,svar(m) / ,array(m) / ,array(row,col)}] ... Specifies the size of string variables and numeric arrays.	✓	✓		3-19
END Stops program execution.	✓			3-2
FOR control var = expr1 TO expr2 [STEP expr3] Begins a FOR-NEXT loop and defines the number of times the loop is executed.	✓			3-21
GOSUB line no. Transfers program control to the first statement of a subroutine.	✓			3-27

F.1 COMMONLY USED BASIC STATEMENTS (Continued)

Formats and Descriptions	S	C	F	Page Ref.
GOTO line no. Transfers program execution to a specified line.	✓			3-30
IF {rel-expr / expr} THEN statement Executes a statement based on whether an expression is true or false.	✓	✓		3-32
INPUT ["string lit",] {var / svar} [{,var / ,svar}]...[;] User inputs data for variables from terminal.	✓	✓		3-5
[LET] {var / mvar} = expr Assigns values or solutions to formulas to a variable.	✓	✓		3-4
NEXT control var Last statement in a FOR-NEXT loop and changes the value of the control variable.	✓			3-21
ON expr {GOTO / GOSUB} line no. [,line]... Transfers program control to a line number whose position in the argument list is computed from an expression.	✓			3-35

F.1 COMMONLY USED BASIC STATEMENTS (Continued)

Formats and Descriptions	S	C	F	Page Ref.
{; / PRINT} [{expr / "string lit" / svar} [{, / ;} {expr / "string lit" / svar}] ...] [{, / ;}]				
Prints specified data.	✓	✓		3-12
RANDOMIZE Reseeds the random number generator.	✓			4-5
READ {var / svar} [{, var / , svar}] ... Reads data from DATA statements.	✓	✓		3-9
REM [message] Inserts explanatory comments within a program.	✓			3-1
RESTORE [line no.] Moves the data element pointer to the beginning of a data list or DATA statement line.	✓	✓		3-11
RETURN Last statement of a subroutine and returns program control to statement following last GOSUB statement executed.	✓			3-27
STOP Stops program execution.	✓			3-3

F.2 ARITHMETIC AND SYSTEM FUNCTIONS

Formats and Descriptions		S	C	F	Page Ref.
ABS(expr)	The absolute value of an expression.			✓	4-8
ATN(expr)	The arctangent of an angle. Result expressed in radians.			✓	4-15
COS(expr)	The cosine of an angle. Angle expressed in radians.			✓	4-13
EXP(expr)	The value of e to the power of an expression.			✓	4-10
FNa(d)	A user function which is defined in a DEF statement and returns a numeric value.			✓	4-17
INT(expr)	The integer value of an expression.			✓	4-7
LOG(expr)	The natural logarithm of an expression.			✓	4-11
RND(expr)	Random number between 0 and 1.			✓	4-3
SGN(expr)	The algebraic sign of an expression.			✓	4-6
SIN(expr)	The sine of an angle. Angle expressed in radians.			✓	4-12
SQR(expr)	The square root of an expression.			✓	4-9

F.2 ARITHMETIC AND SYSTEM FUNCTIONS (Continued)

Formats and Descriptions		S	C	F	Page Ref.
SYS(0)	The time of day (seconds past midnight).			✓	4-16
SYS(1)	The day of the month.			✓	4-16
SYS(2)	The month of the year.			✓	4-16
SYS(3)	The year.			✓	4-16
SYS(4)	The terminal port number (-1 if operator's console).			✓	4-16
SYS(5)	CPU time used in seconds.			✓	4-16
SYS(6)	The number of file I/O statements executed.			✓	4-16
SYS(7)	The error code of the last run-time error.			✓	4-16
SYS(8)	The number of the file most recently opened.			✓	4-16
SYS(9)	Page size.			✓	4-16
SYS(10)	Tab size.			✓	4-16
SYS(11)	Hour of the day.			✓	4-16
SYS(12)	Minutes past last hour.			✓	4-16

F.2 ARITHMETIC AND SYSTEM FUNCTIONS (Continued)

Formats and Descriptions		S	C	F	Page Ref.
SYS(13)	Seconds past last minute.			√	4-16
SYS(14)	Seconds remaining on timed input.			√	4-16
SYS(15)	The constant PI (3.14159).			√	4-16
SYS(16)	The constant e (2.71828).			√	4-16
TAB(expr)	Function used with PRINT for tabulating to a column.			√	3-17
TAN(expr)	The tangent of an angle. Angle expressed in radians.			√	4-14

F.3 STRING FUNCTIONS

Formats and Descriptions	S	C	F	Page Ref.
LEN(svar) Returns the number of characters currently assigned to a string variable.			√	5-8
POS ({svar1 / "string lit 1"} , {svar2 / "string lit 2"} , expr) Locates the position of a substring in a string.			√	5-9
STR$(expr) Converts a numeric expression to its string representation.			√	5-10
VAL ({svar / "string lit"}) Returns decimal representation of a string.			√	5-11

F.4 MATRIX MANIPULATION

Formats and Descriptions		S	C	F	Page Ref.
MAT mvar1 = mvar2	Assigns the dimensions and values of mvar2 to mvar1.	✓	✓		6-2
MAT mvar1 = mvar2 {+ / -} mvar3	Performs matrix addition or subtraction.	✓	✓		6-11
MAT mvar1 = {mvar2 / (expr)} *mvar3	Multiplies a matrix by a numeric expression or another matrix.	✓	✓		6-13
MAT mvar = CON [(row,col)]	Sets the value of each matrix element to one.	✓	✓		6-5
MAT mvar = IDN [(row,col)]	Sets the elements of the major diagonal of a matrix to ones and all other elements to zeros.	✓	✓		6-6
MAT mvar1 = INV (mvar2)	Performs matrix inversion.	✓	✓		6-16
MAT mvar1 = TRN (mvar2)	Transposes matrix mvar2.	✓	✓		6-20
MAT mvar = ZER (row,col)	Sets the value of each matrix element to zero.	✓	✓		6-3
MAT INPUT mvar [(row,col)] [,mvar[(row,col)]]...[;]	Specifies matrices for which the programmer enters data from the terminal when the statement is executed.	✓	✓		6-9

F.4 MATRIX MANIPULATION (Continued)

Formats and Descriptions	S	C	F	Page Ref.
MAT PRINT mvar [{,;} mvar]...[;] Prints the contents of the specified matrices.	✓	✓		6-10
MAT READ mvar [(row,col)] [,mvar[(row,col)]]... Reads data into the specified matrices from the data list defined by a DATA statement(s).	✓	✓		6-8
var = DET(X) Produces the determinant of the last matrix inverted by the INV statement.	✓	✓		6-19

F.5 FILE INPUT AND OUTPUT

Formats and Descriptions	S	C	F	Page Ref.
CLOSE [FILE (file)] Closes an open file or files.	✓	✓		7-8
EOF (file) Returns a +1 if an end of file is detected.			✓	7-20
INPUT FILE (file) {var / svar} [, {var / svar}] ... Reads data in ASCII from a sequential access file.	✓	✓		7-14
MAT INPUT FILE (file),mvar[,mvar]... Reads matrix data in ASCII from a sequential access file.	✓	✓		7-19
MAT PRINT FILE (file),mvar [{, / ;} mvar] ...[;] Outputs matrix file data to an ASCII device.	✓	✓		7-18
MAT READ FILE ({file / file,record}),mvar[,mvar]... Reads matrix data in binary format from a file.	✓	✓		7-16
MAT WRITE FILE ({file / file,record}),mvar[,mvar]... Writes matrix data in binary format to a file.	✓	✓		7-15

F.5 FILE INPUT AND OUTPUT (Continued)

Formats and Descriptions	S	C	F	Page Ref.
OPEN FILE (file,mode), filename [,record size[,filesize]] Opens a file which can then be referenced by other file I/O statements.	✓	✓		7-3
PRINT FILE (file) {expr / var / svar / "string lit"} [{,/;} {expr / var / svar / "string lit"}] ... [{,/;}] Outputs data to an ASCII device.	✓	✓		7-13
READ FILE ({file / file,record}) ,{var / svar} [,{var / svar}] ... Reads data in binary format from a file.	✓	✓		7-11
WRITE FILE ({file / file,record}) ,{expr / var / svar / "string lit"} [,{expr / var / svar / "string lit"}] ... Writes data in binary format to a file.	✓	✓		7-9

F.6 INTERACTIVE SYSTEM COMMANDS

Formats and Descriptions		S	C	F	Page Ref.
BYE	Sign-off command.	✓	✓		8-18
CHATR filename,attributes	Changes file attributes.	✓	✓		8-34
CON	Continues execution of a STOPped program.		✓		8-15
DELETE filename	Deletes a file from the programmer's directory.	✓	✓		8-32
DISK	Prints the number of blocks used and available in the programmer's directory.		✓		8-31
ECHO	Enables echoing of characters at the terminal.	✓	✓		8-27
ENTER filename	Merges the program named into the current program.	✓	✓		8-10
ERASE line nl, line n2	Deletes statements from a program.	✓	✓		8-3
ESC	Enables use of ESC key at the terminal.	✓	✓		8-25
FILES	Prints the filenames in the programmer's directory.		✓		8-28
LIBRARY	Prints the filenames in the library directory.		✓		8-29

F.6 INTERACTIVE SYSTEM COMMANDS (Continued)

Formats and Descriptions		S	C	F	Page Ref.
LIST [line n1 / To line n2 / line n1 {TO, ,} line n2] [filename]	Outputs part or all of the current program to the terminal or other ASCII device.		✓		8-4
LOAD filename	Loads a previously SAVEd program into the program storage area.		✓		8-9
MSG[ΔuserIDΔmessage]	Transmits messages to other users or the operator or cancels NOMSG.		✓		8-21
NEW	Clears the programmer's storage area.	✓	✓		8-2
NOECHO	Inhibits echoing at the programmer's terminal.	✓	✓		8-26
NOESC	Disables ESCape key operation.	✓	✓		8-24
NOMSG	Prevents the reception of messages from other programmers.		✓		8-23
PAGE=*expr*	Sets the right margin of the terminal.	✓	✓		8-19

Formats and Descriptions	S	C	F	Page Ref.
PUNCH [line n1 / To line n2 / line n1 {TO, ,} line n2]				
Outputs part or all of the current program to the terminal punch.		✓		8-6
RENAME oldfilename, newfilename				
Renames files.	✓	✓		8-33
RENUMBER [line n1 / STEP line n2 / line n1 STEP line n2]				
Renumbers statements in the current program.		✓		8-13
RUN [line no. / filename] Executes the current program or another program named by filename.		✓		8-11
SAVE filename Writes the current program into the programmer's directory or to a device in binary format.	✓	✓		8-8
SIZE Provides program and data storage usage information.			✓	8-17
TAB=*expr* Sets the zone spacing for PRINT statements.	✓	✓		8-20
WHATS filename Prints attributes and other information relating to a file.		✓		8-30

F.7 ADVANCED BASIC STATEMENTS AND COMMANDS

Formats and Descriptions		S	C	F	Page Ref.
CALL subr [,expr]...	Calls an assembly language subroutine.	✓			9-20
CHAIN filename [THEN GOTO line no.]	Transfers control to the program named in the statement.	✓			9-18
DELAY=expr	Delays program execution for a specified amount of time.	✓			9-4
ON ERR THEN statement	Directs the program to an error handling routine when an error occurs.	✓			9-2
ON ESC THEN statement	Directs the program to a user handling routine when ESCape is pressed.	✓			9-5
PRINT FILE (file), USING format, expr [,expr]...	Formats output to files.	✓	✓		9-17
PRINT USING format, expr [,expr]...	Formats printed output.	✓	✓		9-7
RETRY	Repeats the statement which caused caused an error.	✓			9-3
TIME=expr	Establishes the time limit for timed input operation.	✓	✓		9-21

Formats and Descriptions	S	C	F	Page Ref.
TINPUT["*string lit*",]{*var* / *svar*} [{,*var* / ,*svar*}] ... [;] Used in conjunction with TIME to set a limit for programmer response.	√	√		9-23

DataGeneral

PROGRAMMING DOCUMENTATION
REMARKS FORM

Document Title	Document No.	Tape No.

SPECIFIC COMMENTS: List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

GENERAL COMMENTS: Also, suggestions for improvement of the Publication.

FROM:

Name	Title	Date

Company Name

Address (No. & Street)	City	State	Zip Code

Form No. 10-24-004

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary If Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE