Prime Computer User Guide

For

BASIC

Interpretive Language

PRIME COMPUTER, INC., 145 PENNSYLVANIA AVENUE, FRAMINGHAM, MA. 01701

MAN 1813

BASIC

INTERPRETIVE LANGUAGE

USER GUIDE

Revision A

May 1975

PRIME
COMPUTER, INC.

[145 Pennsylvania Ave., Framingham, Mass., 01701]

First Printing May 1974
Revision A May 1975

MAN 1813

Prime Computer, Incorporated

145 Pennsylvania Avenue

Framingham, Massachusetts 01701

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FOREWORD

BASIC is easy to learn and easy to use. The rules of form and usage are simple. This manual describes the Prime BASIC language processor and demonstrates how it is used to solve problems and cope with features common to computers. It is suitable for (1) people who know BASIC and want to know what Prime's BASIC is like and (2) experienced programmers. The tyro is advised to supplement this book with a primer on BASIC.

Prime BASIC is an extended subset resembling the BASIC developed at Dartmouth College. It provides users with the ability to write programs and get meaningful results from the computer in a relatively short time. With a few hours of instruction and/or practice, most people can produce worthwhile BASIC programs and obtain useful data from them.

Section 1 describes the structure of a BASIC Program, gives a few general rules about writing BASIC program statements, and tells how to enter BASIC and how to input, edit, and RUN programs.

Section 2 describes in detail how numeric and string data are represented in BASIC, and gives limits of numeric and string data values.

Section 3 describes both numeric and string expressions, expression operators, and expression evaluation.

Section 4 describes the organization, and the input and output of program and data files.

Section 5 describes the statements available in the BASIC language. The function and syntax of each statement is described and examples of each statement are given along with the description.

Section 6 describes the statements available to manipulate matrices and vectors.

Section 7 describes how to interface called FORTRAN or PMA language programs.

Appendix A gives some sample programs using the BASIC language.

Appendix B lists the error messages returned by the BASIC language processor and the definitions of those error messages.

Appendix C is a quick summary of all the features of BASIC.

Appendix D describes a utility program to renumber BASIC programs.

Appendix E describes the memory requirements for various versions of Prime BASIC.

VERSIONS OF BASIC

On the master disk (i.e., that disk supplied to the Prime customer with all the current software), there are three versions of the BASIC language interpreter: BASIC, LBASIC, and DBASIC.

The version named BASIC contains a subset of the BASIC language that does not include MAT functions or PRINT USING functions. It is intended for user's who may have memory limitations or who may only want a simple form of BASIC. LBASIC is a full version of the BASIC interpreter that includes MAT functions and PRINT USING functions. DBASIC is a full version of BASIC that takes advantage of Prime's double-precision floating point arithmetic capabilities.

All versions of BASIC on the master disk may be copied to the users command directory (CMDNC0) or to a user designated UFD and renamed "BASIC" by use of the operating system's FUTIL and CNAME commands, (Refer to the Disk and Virtual Memory Operating Systems User Guide).

DOUBLE PRECISION BASIC

Prime BASIC has been revised to include double precision floating point representation. BASIC with double precision floating point is implemented using floating point hardware; thus, coding that references floating point operations is both in line and efficient.

All constants, variables, and array elements are represented in floating point format with a 48-bit mantissa and a 16-bit exponent. This representation allows numbers to have an accuracy up to 14.2 decimal places. With double precision floating point, it is possible to represent a number up to:

9,999,999,999,999

or a dollar sum up to:

$99,999,999,999.99

without resorting to the use of scientific format.

To use the double precision version of BASIC, type the command:

DBASIC

Use of this version of BASIC, other than for the extensions outlined, is identical to the use of BASIC described in this manual.

RELATED PUBLICATIONS

The following Prime documents should be available for reference:

Disk and Virtual Memory Operating Systems User Guide

Program Development Software User Guide

SECTION 1

STRUCTURE OF A BASIC PROGRAM

STATEMENTS

A BASIC program consists of a series of sequentially ordered statements.

Statement Format

Each statement is preceded by an integer called the statement number. This number serves as both a statement sequence number as well as a line identifier. An example of a BASIC statement is:

```
100  PRINT 'AARDVARK'
```

Each statement must be contained on one line. The length of a line is dependent on the number of characters that can be typed before a carriage return is needed to prevent the line from overflowing. BASIC will accept lines up to 120 characters in length.

Statement Execution

When a program written in the BASIC language is run, the statements are executed in order of statement number (unless a statement such as GOTO affects the normal order).

ENTERING BASIC

To enter BASIC from operating system command level, type:

```
BASIC
```

The system then replies:

```
GO
>
```

The character '>' indicates that the BASIC processor is awaiting a command, and is printed as a prompt.

MODES OF OPERATION

The Prime BASIC language processor consists of a command processor, a statement editor and a BASIC language interpreter.

After entering BASIC from operating system command level, GO is typed. The user may:

1. Input, edit, and RUN programs written in the BASIC language (conversational mode);

2. Execute existing programs written in BASIC language and stored on disk or paper tape (batch mode);

3. Execute BASIC statements as they are typed at the terminal (immediate mode).

CONVERSATIONAL MODE

Entering Program Statements

To enter a statement, type the statement number followed by the body of the statement. All statements must be terminated by a carriage return.

Statement Numbers: Statement numbers are integers that range from 1 to 9999. They do not have to be in cardinal sequence (i.e., 1, 2, 3...n-1,n), but they must be in an ordered sequence (e.g., 10, 12, 15, 20...n). It is recommended that statements be numbered by increments of 10 (100, 110, 120, 130...). Then, if a statement must be inserted between 10 and 20, for example; it can be numbered 15, and it is inserted between 10 and 20.

For example:

```
110   PRINT  'NAME', N$
120   PRINT  'ADDRESS', A$
130   PRINT  'CITY', C$
```

To insert lines between 110 and 120, and 120 and 130, in order to make the output more readable, the user need only type:

```
115   PRINT
125   PRINT
```

at his terminal. The resulting program sequence is as follows:

```
110  PRINT  'NAME', N$
115  PRINT
120  PRINT  'ADDRESS', A$
125  PRINT
130  PRINT  'CITY',  C$
```

Body of Statement: In the conversational mode, each statement starts after its statement number with a full or partial English word. This word denotes the type of the statement.

Examples of BASIC statements are:

```
100  REM   THIS IS A REMARK
110  LET  X = 2
120  PRINT  X
```

Blanks: Blanks (spaces) have no significance except in string constants. Generally, spaces are used to make the program more readable. For example:

```
110  LET  X  =  3.14
110   LET   X = 3.14
110  LETX=3.14
```

are equivalent. Thus, BASIC statements are free-formatted and the user may employ spaces at will to format BASIC program text.

Special Characters: The following characters have special meaning:

"	removes the character previously typed.
?	removes all previous characters on a line.
CARRIAGE RETURN	Terminates a source statement.

This convention is consistent with the operating system and the system Editor, (refer to the Disk and Virtual Memory Operating System User Guide and the Program Development Software User Guide).

Storage of Statements

When the CARRIAGE RETURN is received by the BASIC language processor, the statement is stored into the program storage area. Statements may be entered in any order, but their execution occurs in the order of their statement number.

Replacing a Statement

If a statement is entered with the same number as a statement already in the program storage area; the previous statement is removed, and the new statement is placed in the storage area instead.

Example:

Existing statement is:

```
110  LET  X1  =  Y**2
```

Assume a new statement is typed as follows:

```
110  LET  X1  =  Y↑2
```

The second statement numbered 110 replaces the first statement.

Deleting a Statement

To remove an existing statement without replacing it, type the statement number followed by a CARRIAGE RETURN. Example:

```
110
```

deletes statement numbered 110.

Summary of BASIC Program Editing Procedures

To input a statement, type:

unused statement number, followed by statement, followed by CARRIAGE RETURN.

To insert a statement, type:

a statement using a statement number between the two statements surrounding the insertion.

To replace a statement, type:

a new statement with a statement number that is identical with the number of the statement to be replaced.

To delete a statement, type:

the statement number, followed by a CARRIAGE RETURN.

EXECUTING A PROGRAM

To run all BASIC statements in the program storage area, the user types:

```
RUN
```

This causes the BASIC language processor to interpret and execute the program comprised of the statements in the program storage area.

BATCH MODE

In addition to input from a terminal, statements may be input to the BASIC processor from source files on disk, or from off-line storage devices such as paper tape, magnetic tape, or cards.

Data to be processed during RUN time may come either from the program itself (DATA statements), from the terminal (via use of INPUT statements), or from files on disk. Output data from a program written in BASIC may either be printed at the terminal or placed in a file on the disk.

Batch mode requires the reading and writing of files via the use of the LOAD and FILE commands.

After a BASIC program is written, it may be saved in the User File Directory (UFD) via a FILE command. (For information on the UFD, see the Prime Disk and Virtual Memory Operating Systems User Guide.) The syntax of the FILE command internal to BASIC is:

```
FILE  'FILENAME'

      or

FILE  'FILENAME', S1, S2
```

where FILENAME is the symbolic name of the file to be created or updated enclosed in single quotes. The single quotes are delimiters necessary to BASIC and are not part of the file name. The file FILENAME is updated; however, the contents of the BASIC program storage area remain unchanged. FILENAME may also be a parenthesized device name (see the DEFINE FILE statement discussion in Section 5). The optional argument S1 specifies the first statement number of the BASIC program to be filed. If S1 is omitted, its default value is 1. The optional argument S2 is the last line to be filed. If S2 is omitted, its default value is 9999. All statements having statement numbers in the inclusive range S1 through S2 are output to the specified file or device.

Examples of FILE Command

```
FILE 'RANDXX'
```

creates a file named RANDXX in the UFD.

```
FILE '(PTP)', 100, 200
```

creates a file for output to the paper tape punch consisting of all the statements in the program storage area with statement numbers between 100 and 200 inclusive. The contents of the program storage area remain unchanged.

Loading and Running Programs

To load and run a BASIC program that has been previously edited and saved in a file, the user loads the program by using the LOAD command, and executes the program by issuing a RUN command immediately after issuing the LOAD command.

The syntax of the LOAD command is:

```
LOAD  'FILENAME'

or

LOAD  'FILENAME', S1
```

where FILENAME is the name of a file in the UFD or a symbolic device specification and the single quotes are delimiters required by BASIC. The optional argument S1 is a statement number specifying that all statements in the loaded source files are to be biased by the specified statement number value, in order to avoid conflict with any program already loaded. If S1 is omitted, statements in the program storage area are numbered the same as the corresponding statements in the file.

The RUN command may have been written as the last line of the source file by use of the system editor. In this case, the initial LOAD command causes the program to be both loaded and run.

Examples of LOAD Command

The command line:

```
LOAD  'RANDXX'
```

loads the previously saved file RANDXX into the program storage area.

```
LOAD '(PTR)', 1000
```

loads a file from the paper tape reader, and starts numbering the stored statements at statement number 1000.

After the program, or programs are stored using the LOAD command, the user executes all statements stored by typing:

```
RUN
```

The following is an annotated example of some trivial programs written in BASIC. In shows simple editing of a series of program statements in conversational mode and the loading and running of programs using BATCH mode concepts. The use of the BASIC FILE, NEW, LIST and LOAD commands is also illustrated. User input is underlined.

```
OK, BASIC
GO
>10 REM START                    } BASIC is invoked and a simple
>20 PRINT 'AARDVAARK'            } program is typed in by the user
>30 END
>FILE 'AARD'  <-- To save this program as a file
>NEW  <-- To clear program storage area
>30 PRINT 'SYZYGY'
>40 END  <-- Typing a new program
>FILE 'SYZYGY'
>QUIT  <-- To exit from BASIC

OK, ...
```

```
OK, BASIC
GO
>LOAD 'AARD'                     } At a later time BASIC is entered and
>LOAD 'SYZYGY'                   } the filed programs are loaded
>LIST  <-- To list the contents of the program storage area.
  10  REM START
  20  PRINT 'AARDVAARK'
  30  PRINT 'SYZYGY'
  40  END
>RUN  <-- To execute program
AARDVAARK                        } Output from user program
SYZYGY                           }

END AT LINE 40
>
```

IMMEDIATE MODE

Immediate mode allows a user to type BASIC statements with no statement number and thereby obtain immediate results. Such statements are not stored in the program storage area. For example:

```
PRINT 'XYZ'
```

causes the string XYZ to be printed at the terminal.

The immediate mode capability gives the user a super-calculator with a rich choice of functions, automatic decimal point handling, and up to 286 variables, as well as arrays available for partial answer storage.

One use of immediate mode is to use the BASIC subsystem as a desk calculator. For example:

```
X = 256*12

PRINT X
```

returns the product of 256 and 12.

The PRINT statement is a particularly useful immediate mode command. For example:

```
LET X1 = 1.05

PRINT SIN (X1* 3.14959/180)
```

causes the appropriate value of the SIN function to be issued.

The immediate mode is useful at times for debugging programs written in BASIC. For example, if the user has made use of the BREAK statement (Section 5) to halt a program at some point, typing:

```
PRINT J2
```

prints the value of the variable J2 at the point that the execution of the program was interrupted.

Similarly, it is possible to use the PRINT statement to print the value of any and all variables at a point of interruption.

COMMANDS

The BASIC language processor provides a number of commands to be used with the operating system and to initialize storage areas. Of these, use of RUN, FILE and LOAD have been previously discussed.

These commands are usually executed in immediate mode, but they may be part of a program statement.

The syntax and function of system commands are described in the following paragraphs:

LOAD COMMAND

Syntax

LOAD 'FILENAME'

or

LOAD 'FILENAME', S1

'FILENAME' - is a string constant that specifies the file to be created (or parenthesized device specified). The single quotes are delimiters required by BASIC.

S1 - is a relocation constant that is added to every statement number in the program, written in BASIC, to be loaded.

Function

The specified file (of BASIC Source Statements) is loaded into the BASIC program storage area.

The loaded program is merged with any program already loaded. For examples, see the previous section entitled "Examples of Load Command".

FILE COMMAND

Syntax

FILE 'FILENAME'

or

FILE 'FILENAME', S1

or

FILE 'FILENAME', S1, S2

'FILENAME' - (is the same as described for LOAD, above.)

S1 (optional) = first line to be filed (default = 1).

S2 (optional) = last line to be filed (default = 9999).

Function

All statements whose statement numbers are in the inclusive range S1 through S2 are output to the specified disk file or output device. Output is in the order of their statement numbers. Example.

```
FILE 'NEWPRO'
```

LIST COMMAND

Syntax

```
LIST

or

LIST S1

or

LIST S1, S2
```

S1 = first line to be listed (default = 1).

S2 = last line to be listed (default = 9999).

Function

The LIST command prints output at the terminal. The LIST command provides a means to print all or part of the previously edited statements for the user's inspection.

Examples:

```
LIST

LIST 100, 250
```

RUN COMMAND

Syntax

```
RUN

or

RUN S1
```

S1 = statement number specifying the first statement to be executed (default is the first statement in the program).

Function

RUN clears all variables, allocates arrays from DATA statements, and starts program execution.

NEW COMMAND

Syntax

NEW

Function

The NEW command deletes all existing program statements and de-allocates all arrays and variables.

CLEAR COMMAND

Syntax

CLEAR

Function

The CLEAR command de-allocates all arrays and variables. Any existing statements are not deleted.

CONTINUE COMMAND

Syntax

CONTINUE

Function

The CONTINUE command restarts program execution at the point that it was last interrupted by a BREAK, STOP or END statement.

RESTARTING BASIC

Restarting from DOS/VM

The user may desire to QUIT from running a BASIC program (e.g., to avoid printing unwanted output), and then return to running under control of BASIC. Naturally, it is desirable not to lose any information in the program storage area or cause any unspecified operations. For DOS/VM, the correct manner to achieve this result is to type the following sequence of system command lines:

CONTROL-P	(Quit by pressing terminal CTL and P Keys simultaneously).
START 1002	

Restarting from DOS

Under DOS, to QUIT from running a BASIC program, momentarily set Sense Switch 1. The running program is interrupted and contro; returns to BASIC command mode.

Return from INPUT

The user may type the sequence:

CONTROL-C	To return from BASIC INPUT statement execution to conversational mode (Refer to Section 5).

ERROR MESSAGES

Statements are syntactically checked as they are entered. Errors that can only be detected within the context of the entire program are detected at run time. An example of a syntax error is:

```
100  PRINT       'SUM OF A & B IS: X
```

The closing ' mark is missing and this would be detected immediately upon entry. An example of a context error is an undefined statement number in a GOTO statement.

If an error is detected during statement input, a two-line error is printed at the terminal. The first line is the source statement in error. The second line consists of first, a vertical arrow positioned under the last character that BASIC examined before detecting the error, and then a two-character error code. These codes are listed as source (S) errors in the table in Appendix B.

Errors detected during program input cause the line in error to be removed from the program.

During program exectuion (RUN time), detected errors cause a one-line message to be printed as follows:

```
ERROR  XX  LINE  385
```

where XX is the error code. These codes are listed as execution (E) errors in the table in Appendix B.

Errors detected during program execution also cause a pause to occur. Typing:

```
CONTINUE
```

causes processing to continue with the next statement.

SECTION 2

TYPES OF DATA

Two types of data are supported by Prime BASIC: numeric and string. BASIC allows constants and variables of both types.

NUMERIC VALUES

A numeric value is a floating point number. Depending on the version of BASIC being used, it may be single or double precision.

A numeric constant is written as a signed decimal number. It may contain a decimal point, and it may be followed by an exponent.

The exponent field is optional and is written as the letter E followed by an optionally signed decimal integer.

If the decimal point is omitted, it is assumed to be located immediately to the right of the last significant digit (right-most digit).

If the sign of either the numeric constant or the decimal integer exponent is omitted, it is assumed to be positive.

Examples:

12	
1.2	
-6.666	
-7	
2.5E-2	(.025)
2.5E+12	(2.5 * $(10)^{12}$)
-7.3E-2	(-.073)
5E5	(500000)

Range of Numeric Values

For single-precision values; all constants, variables and array elements are represented in floating point format with a 24-bit mantissa and an 8-bit exponent. This representation allows numbers to have accuracy up to 6.2 decimal digits, and the exponent of a single-precision numeric value may range between -38 and +38. (10 to the -38 power, or 10 to the +38 power).

With single-precision format, it is possible to represent a number up to: 999,999 or a dollar sum up to: $9,999.99 without resorting to scientific format.

For double-precision values; all constants, variables, and array elements are represented in floating point format with a 48-bit mantissa and a 16-bit exponent. This representation allows a number to have an accuracy up to 14.2 decimal places.

With double-precision floating point, it is possible to represent a number up to: 9,999,999,999,999 or a dollar sum up to: $99,999,999,999.99.

STRING VALUES

A string value is a string of ASCII characters.

A string constant is written as a set of 0 or more contiguous ASCII characters enclosed in delimiting single quotation marks (or apostrophes). A string constant can contain any ASCII character except: CARRIAGE RETURN, ?, or ". The maximum length (number of characters) of a string value is a function of the line size of the terminal or upon the available memory. Generally, this is large enough to be of no problem to the user. It is suggested that for convenience no string be greater than 80 characters.

Examples:

```
'THIS IS A CHARACTER STRING CONSTANT'

'DATE/TIME/YEAR'

''    (null string)

'12345'
```

SCALAR VARIABLES

A scalar variable is implicitly defined when it is used in a BASIC program. The type of scalar variable (i.e., numeric or string) is determined by the form of the variable name.

Numeric Scalar Variables

The name of a numeric scalar variable is a single letter (A-Z), or it is a single letter (A-Z) followed by a single digit (0-9). Each variable represents a single numeric value; there are 286 possible numeric scalar variables. A numeric scalar variable is initialized automatically to 0 at the start of the BASIC program that defines it.

Examples of Numeric Scalars:

```
X

A1

G3
```

Example of Use of Numeric Scalars:

```
20  LET  C1  =  3.14157

22  LET  X   =  C1*2
```

String Scalar Variables

The name of a string scalar variable consists of a single letter followed by a dollar sign. A string scalar variable represents a character string of variable length. String variables are initialized to a null (zero length) string at the start of the BASIC program that defines it. The length of a string variable is automatically set to the length of the string that is assigned to it.

Example of String Scalar Variable:

```
B$
```

Example of Use:

```
100  LET B$ = 'BALANCE IS:'
```

ARRAY VARIABLES

An array is an ordered set of values. All elements of an array (array variables) have the same data type (i.e., either numeric or string). The elements of an array are stored in contiguous locations in storage and are referenced by an array subscript. Arrays are stored in column major order.

An array name is represented by a single letter followed by the parenthesized list of one or two bounds.

An array element is designated by an array subscript that is either one number (bound) in parentheses (one-dimension), or two numbers (bounds) in parentheses and separated by commas (two-dimensions). An array with one-dimension may be operated on as a vector; with two dimensions, it may also be operated on as a matrix (See Section 6).

Examples:

```
A (6)

A (2, 3)
```

Conceptually, the array A (2,3) is:

A (0,0)	A (1,0)	A (2,0)
A (0,1)	A (1,01)	A (2,1)
A (0,2)	A (1,2)	A (2,2)
A (0,3)	A (1,3)	A (2,3)

Table 2-1. Example Array A (2,3)

Logically, the array A (2,3) maps into storage as shown in the following table:

Relative Location	Element
0001	A (0,0)
0002	A (1,0)
0003	A (2,0)
0004	A (3,0)
0005	A (0,1)
0006	A (1,1)
0007	A (2,1)
0008	A (3,1)
0009	A (0,2)
00010	A (1,2)
00011	A (2,2)
00012	A (3,2)

Table 2-2. Array Mapped into Memory

Array Declaration

An array can be explicitly defined in a DIM statement, or implicitly defined by its use in the program.

DIM statements, if used, may appear anywhere in the program, since BASIC locates and interprets all DIM statements before execution starts.

Examples:

```
DIM A (5)
```

defines a one-dimensional array of 6 locations A, A (0) through A (5).

```
DIM A (2, 3)
```

defines a two-dimensional array of 3 columns and 2 rows, A (0,0 through A (2,3).

NOTE: The entire chart shown in Table 2-1 is the array specified by the DIM statement, DIM A (2,3). Those elements of the array that do not have zero subscripts (e.g., A (1,2); A (1,3); A (2,1); A (2,2); A (2,3) define the matrix A. This matrix may be manipulated via the MAT statements described in Section 6.

If the DIM statement is omitted (i.e., an array is undeclared), the array dimensions are established in any MAT statement encountered; otherwise the array is either a one-dimensional array of no more than 10 elements (e.g., A(10)), or a two-dimensional array of bounds 10 by 10 (e.g., A(10,10)), depending on how the array is referenced.

Use of an array in a MAT statement can cause the array to be defined either as a vector or matrix depending on the other arrays used in the statement (refer to Section 6).

Array Bounds, Default Bounds, and Storage Allocation

The original bounds of an array are established by the DIM statement that defines the array, by the first MAT statement that references the array, or the implicit value ((10) or (10,10)). The original bounds of an array specify the total amount of storage allocated for the array. The MAT statement can reduce the size of an array, but the MAT statement cannot increase the size of the array beyond that of the original definition. Although the dimensions of an array may be changed, the storage allocation for the array does not change during execution of the BASIC program.

Array Element References

Numeric Arrays: The name of a numeric array is a single letter (A-Z). When a single element of an array is initialized to any value, the remaining elements of numeric arrays are initialized to 0.

String Arrays: The name of a string array is a single letter followed by a dollar sign, $. The elements of a string array are variable-length character strings. These character strings may all be of different lengths. Elements of a string array are initialized to a null value when the array is established.

A reference to an array element consists of the array name followed by a parenthesized list of one or two subscripts; i.e., A (S1) or A (S1, S2), where A is the array and S1 and S2 are positive numeric expressions (see Section 3 for a discussion of expressions).

Examples of Numeric Arrays:

```
A(5)

A(2,4)

A(K, J)          where K and J are numeric scalar
                 variables

A(I+1, J/2)

A(I+J, 3*K-2)
```

If the value of a subscript expression is fractional, the value of the subscript is truncated to an integer before it is used to locate the specified array element.

The value of any array subscript expression must be within the range of the corresponding array dimension.

Examples of String Arrays:

```
A$(5)

A$(I+1, 3*K-2/J)

A$(A (I) /4)
```

Relationship of Names

A string variable and a string array may have the same name in a program. Likewise, a numeric variable and a numeric array may have the same name. However, these names all refer to entirely different entities. The context in which the name is used is the determining factor. For example:

```
10  B$  =  'BBBBB'

20  DIM  B$ (7)

25  B = 2

30  DIM B (7)
```

are different variables even though the names are apparently the same. B$ references a string scalar variable; B$ (7) references a string array of 8 elements (0-7); B references a numeric scalar variable and B (7) references a numeric array.

SECTION 3

EXPRESSIONS AND FUNCTIONS

The first part of this section describes the arithmetic and string expressions that may be constructed in the Prime BASIC language. The second part describes functions, both user defined functions, and system functions provided by BASIC, such as SIN, LOG, etc.

EXPRESSIONS

BASIC expressions are constructed from operators and operands. An operand may be a constant, a scalar variable, subscripted array element, or a function reference.

Operators that require two operands are called binary operators. Operators that require one operand are called unary operators.

BASIC defines two types of expressions: numeric and string. Numeric operands must not be used with string operators and string operands must not be used with numeric operators. There is no conversion between numeric and string values. The user must define explicit conversion functions to convert from one data type to another.

Numeric Expressions

BASIC defines two unary operators and five binary operators that operate on numeric operands to produce a numeric value.

Operator	Meaning	Example
+	unary plus	+I
-	unary minus	-I
+	addition	I + J
-	subtraction	I - J
*	multiplication	I * J
/	division	I / J
↑	exponentiation	I ↑ 2

Table 3-1. Numeric Operators

The operators listed in Table 3-1 have their normal arithmetic meaning. The operations are performed in floating-point arithmetic.

The user is cautioned that if he uses the system editor to create a BASIC source program, then escape conventions must be observed to produce some of the symbols desired. For example, using the system editor, the exponentiation operator (↑) must be escaped by typing a double vertical arrow (↑↑).

Order of Expression Evaluation

A numeric expression is evaluated in the order of operator priority. This is determined by rules of precedence in the BASIC langauge processor. These rules of precedence are:

Precedence	Operator
3	↑
2	unary (+,-), *, /
1	+, -

Operators with higher precedence are evaluated before operators with lower precedence.

Operators with equal precedence are evaluated from left to right.

Example:

```
A + B - C * D *  E ↑ F ↑ G
```

is interpreted as:

```
(A + B) - ((C * D) * ((E ↑ F) ↑ G))
```

Use of Parentheses

Parentheses can be used to control the order of expression evaluation. The operation inside of the parentheses is evaluated first.

Example:

```
(A + B) / 2
```

The addition, A + B, is performed, then the division by 2 is performed, even though / has higher precedence than binary +.

String Expressions

String expressions in BASIC are constructed using the concatenation operator (+). This operator combines two string values to produce

a string having a value of the characters of the first string immediately followed by the characters in the second string.

Examples:

```
A$ + B$

'HELLO' + U$ + 'WELCOME TO PRIME DOS VM'

'ABC' + B$

X $ (I-1) + 'Q1' + S$
```

Relational Expressions

BASIC defines six relational operators that may be used in either numeric or string expressions, as long as data types are not mixed. The relational operators are shown in the following table:

Operator	Meaning	Examples	
<	less than	X < Y	X$ <Y$
>	greater than	X1 > Y1	A$ >B$
=	equal	I = J1	C$ = D$
<=	less than or equal	J2 <= J3	A$ <= B$ + C$
=<	less than or equal	J2 =< J3	A$ =< Y$
>=	greater than or equal	Z >= 10	A$ >= C$
=>	greater than or equal	10 => Q1	C$ => B$
<>	not equal	D <> 1	A$ <> ''
><	not equal	A1 >< A2 + A3	A$ >< B$

Table 3-2. Relational Operators

Examples of Relational Expression Use

```
20  IF  SIN  (ABS (K - 3.14) - 1) = (I+1) - 1 THEN 200
30  IF  S$ <> 'T'  THEN  450
```

Evaluation of Relational Expressions

The relational expressions are true if the expressions satisfy the given expression. Examples:

```
120  IF  X =< Y   THEN 900

150  IF  B$ = 'END' THEN 9999

160  IF  B$ >A$  THEN 120
```

String Values in Relational Expressions

When string values are compared in relational expressions, character ordering is determined by ASCII code. If the strings being compared are of different lengths, the shorter of the two strings is extended by adding spaces to the right until the strings are of the same length; then, the strings are compared. Use of string values in relational expressions are given in statements 150 and 160 in the previous set of examples.

FUNCTIONS

BASIC provides system functions and allows the user to provide user-defined functions. A function reference consists of a function name followed by a parenthesized arguement list containing one or more arguments. Function arguments are evaluated before the function is evaluated.

Arguments used in a function reference must match the number and data type of arguments expected by the function.

Function references are evaluated at the point that their value is required. They do not affect the order of operator evaluation.

System Functions

The following list gives the numeric and string functions provided as system functions by the BASIC language processor. In all of the descriptions in the list, X represents any numeric expression, I and J represent any integers, and A$ represents any string expression.

SIN(X)	computes the sine of X, X expressed in radians
COS(X)	computes the cosine of X, X expressed in radians
TAN(X)	computes the tangent of X, X expressed in radians
ATN(X)	computes the arctangent of X, result is in radians
LOG(X)	computes the natural logarithm (base e) of X
EXP(X)	computes e raised to the X power
SQR(X)	computes the square root of X
ABS(X)	computes the absolute value of X
SGN(X)	returns a value based on the sign of X as follows: X < 0 SGN(X) = -1 X = 0 SGN(X) = 0 X > 0 SGN(X) = 1
INT(X)	If X >= 0, returns the greatest integer >= X. If X < 0, returns the least integer >= X.
RND(X)	If X < 0, uses X to initialize the random number generator, and returns X as the function value. If X > 0, uses X to initialize the random number generator, and returns a value in the range zero to one. If X = 0, returns a random number in the range zero < result < 1.
LEN(A$)	returns the length (number of characters) of the string A$.
SUB(A$,I,J) or SUB(A$,I)	returns a substring that is composed of characters I-J of string A$. If J is not specified, the result is a one character substring consisting of character I of string A$.

Examples of Use of System Functions

INT: One use of the INT function is to round numbers. Example:

```
INT (2.9 + .5) = INT (3.4) = 3
```

The INT function can also be used to round any specific numeric value to any specific number of decimal places. Examples:

```
INT  (10*X1 + .5) /10
```

rounds X1 to 1 decimal place.

```
INT  (100 * X1 + .5) /100
```

rounds X1 to 2 decimal places.

RND: To produce twenty three-digit random integers, edit and run the following BASIC program:

```
10  REM PROGRAM TO PRINT RANDOM NUMBERS OF 3-DIGITS OR LESS.
20  FOR I=1 TO 20
30  LET L=RND(0)
35  LET L1=INT(L*1000)
40  PRINT L1
50  NEXT I
60  END
```

EXAMPLE OF OUTPUT

```
RUN
211
852
801
716
673
176
535
507
358
168
373
309
73
266
478
61
645
906
212
699

END AT LINE 60
```

The following example is a program that illustrates a use of each of the system functions previously described; it is followed by sample output so the user can get an idea of the results from using the system functions.

```
100 REM EXAMPLE TO SHOW USE OF SYSTEM FUNCTIONS
110 REM
120 REM MLG 11-29-74
130 REM
140 LET V = .01745
150 REM 1 DEGREE IN RADIANS
160 W = .52359
170 X = .78540
180 Y = 1.04719
190 Z = 1.57079
200 REM W,X,Y,Z EQUIVALANTS 30,45,60,90 DEGREES RESPECTIVELY
210 REM
220 REM TRIGONEMETRIC FUNCTIONS CALCULATIONS
230 S1 = SIN (V)
240 S2 = SIN (W)
250 S3 = SIN (X)
260 S4 = SIN (Y)
270 S5 = SIN (Z)
280 C1 = COS (V)
290 C2 = COS (W)
300 C3 = COS (X)
310 C4 = COS (Y)
320 C5 = COS (Z)
330 T1 = TAN(V)
340 T2 = TAN(W)
350 T3 = TAN(X)
360 T4 = TAN(Y)
370 T5 = TAN(Z)
380 A1 = ATN(T1)
390 A2 = ATN(T2)
400 A3 = ATN(T3)
410 A4 = ATN(T4)
420 A5 = ATN(T5)
430 PRINT 'DEGREES','SIN','COS','TAN','ARCTAN'
440 PRINT
450 PRINT 1,S1,C1,T1,A1
460 PRINT 30,S2,C2,T2,A2
470 PRINT 45,S3,C3,T3,A3
```

```
480 PRINT 60,S4,C4,T4,A4
490 PRINT 90,S5,C5,T5,A5
500 REM
510 REM ARITHMETIC FUNCTIONS (LOG ETC)
520 REM
530 X = 7.50
540 L = LOG(X)
550 E = EXP(X)
560 Q = SQR(X)
570 A = ABS(X)
580 I = INT(X)
590 P = SGN(X)
600 PRINT
610 PRINT
620 PRINT 'NUMBER =',X
630 PRINT 'LOG(X)',L
640 PRINT 'EXP',E
650 PRINT 'SQUARE ROOT',Q
660 PRINT
670 PRINT 'ABS(X)','INT(X)','SIGN(X)'
680 PRINT
690 PRINT A,I,P
700 PRINT
710 PRINT
720 REM RANDOM NUMBER FUNCTIONS
730 REM
740 PRINT 'RANDOM NUMBER FUNCTIONS'
750 PRINT
760 PRINT 'RND(0)','RND(N)','RND(-N)'
770 PRINT
780 Z1 = RND(0)
790 Z2 = RND(1)
800 Z3 = RND(-1)
810 PRINT Z1,Z2,Z3
820 PRINT
830 REM STRING FUNCTIONS
840 REM
850 X$ = 'EVALUATION OF STRING EXPRESSIONS'
860 PRINT 'VALUE OF A GIVEN STRING:'
870 PRINT
880 PRINT X$
890 PRINT
900 L1 = LEN(X$)
910 PRINT 'LENGTH OF STRING:'
920 PRINT L1
930 PRINT
940 PRINT 'SUBSTRING POSITIONS 21-31:'
950 B$ = SUB(X$,21,31)
960 PRINT B$
970 PRINT
980 END
```

Sample Output:

```
>>DEGREES        SIN             COS             TAN             ARCTAN


1               1.74491E-02     .999847         1.74518E-02     1.745E-02
30              .499992         .86603          .577338         .52359
45              .707108         .707106         1               .785399
60              .866021         .500007         1.73202         1.04719
90              1               6.74112E-06     148343          1.57079

NUMBER =        7.5
LOG(X)          2.0149
EXP             1808.04
SQUARE ROOT     2.73861

ABS(X)          INT(X)          SIGN(X)

7.5             7               1

RANDOM NUMBER FUNCTIONS

RND(0)          RND(N)          RND(-N)

.211273         1               .211273
```

Sample Output: (Cont)

```
VALUE OF A GIVEN STRING:

EVALUATION OF STRING EXPRESSIONS

LENGTH OF STRING:
32

SUBSTRING POSITIONS 21-31:
 EXPRESSION


END AT LINE 980
```

User Functions

In addition to the system functions, BASIC allows the user to define functions. These functions are local to the BASIC program that contains them.

The name of a user-defined numeric function consists of the letters FN followed by a single letter.

Example:

```
FNA (X)
```

A reference to a user defined function consists of the name of the function followed by a parenthesized argument expression.

A user defined function is defined by use of the DEF statement (see Section 5). For example:

```
120  DEF  FNA (X2) = 3.14 * X1+2
```

A user defined function reference may be included as an operand in an expression. Example:

```
170  LET A1 = 3.14 / FNA (X1)
```

The argument of a user-defined function may be an arithmetic expression. The expression in the function reference argument is evaluated, and the value of the expression substituted for the argument in the function definition. For example:

```
180  LET A1 = 3.14 * FNA (X1 + COS (B(3,4))
```

SECTION 4

FILES

DEFINITION

A BASIC file is a set of data external to the BASIC program. A file is known to the operating system by its association with an input/output device. The data in a BASIC file are organized into sequential records. The contents of a file are made available to the program by the execution of input/output statements that transmit data between the file and the program.

The PRIME BASIC allows the user to create and use both program and data files.

PROGRAM FILES

A program file may be created by using the operating system editor (ED or FILED), to create a file consisting of sequentially ordered BASIC statements. For details, refer to the Prime Disk and Virtual Memory Operating System manual, and the Program Development System User Guide.

Generally, a BASIC program file is created by first, editing a program in conversational mode, as described in Section 1; then, using the FILE command, described in Section 1, to write the program file in storage. For example:

```
FILE  'RANDII'
```

stores the contents of the program storage area in a file on disk named RANDII.

After a program file has been created, it may be loaded and executed by entering BASIC and typing the LOAD and RUN commands. For example:

```
BASIC
GO
>LOAD 'RANDII'
>RUN
```

The word GO and the > character before the LOAD and RUN commands are responses printed by the BASIC language processor.

DATA FILES

Data files for input to a BASIC program are created by using the operating system editor (ED or FILED) to create files or by using other BASIC or FORTRAN programs (Refer to Section 5, DEFINE FILE, for a description of possible file formats).

An ASCII file, the most used type of file, is a string of ASCII characters organized into lines followed by a CARRIAGE RETURN. A line consists of a contiguous string of characters between a CARRIAGE RETURN character and the next CARRIAGE RETURN character in the file. The length of a record in a file can be up to 72 characters, including the commas and the CARRIAGE RETURN. Each data item in the file must be separated from the other items by a comma.

Data files are read, manipulated, and written, by DEFINE FILE, DEFINE READ FILE, READ, REWIND and WRITE statements within any programs written in the BASIC language, that reference data files.

File Names

The name of a file stored on disk is a string of six ASCII characters enclosed in single quotes. This string is used by the BASIC interpreter to locate the file. An example of a file name is:

```
'RAND1X'
```

A file name may also be a parenthesized device name (see Section 5).

File Numbers

A BASIC program refers to files by means of a logical file number. The range of file numbers is between 1 and 8 inclusive. The correspondence between a file name and a file number is established by the DEFINE FILE (or DEFINE READ FILE) statement. A file is considered to be open if it is currently assigned a file number; otherwise, it is considered to be closed.

A DEFINE FILE statement in a BASIC program causes an attempt to locate the specified file. No error message is printed if the file cannot be located, unless the file was referenced in a DEFINE READ file statement, in this case, an error message is printed. However, even if the absence of a specified file is not detected, subsequent statements that reference the file may produce an error message.

A file remains open until it is closed. A file can be closed when:

1. control returns from a BASIC program to an operating system (either normally or abnormally). All files opened by that program are then closed.

2. a file is closed if its file number is used in a subsequent DEFINE FILE statement.

File Expressions

The user can write an expression in a DEFINE FILE statement that is evaluated to form a file number. The value of this expression is truncated to an integer if it is a non-integer.

SECTION 5

STATEMENTS

This section describes all the BASIC statements implemented by the Prime BASIC language processor except for the array manipulation statements. These are described in Section 6.

In all the examples shown in this section and Section 6, the response character, >, and the INPUT statement prompt character ! are not shown unless deemed necessary for the purposes of the example.

Certain statements are only available on larger memory configurations (16K memory), notably the MAT and PRINT USING statements. Table 5-1 is a list of configurations and the extent of the BASIC implementation on those configurations. Appendix F gives further details with regard to memory mapping and memory sizes.

Version of BASIC	Memory Size
BASIC without MAT or PRINT USING	16K
BASIC with MAT statements	32K
BASIC with PRINT USING statement	32K
BASIC with both MAT and PRINT USING statements	32K
BASIC with Double Precision	32K

Table 5-1. List of Configurations and BASIC System Statement Availability

BREAK

The BREAK statement selectively enables or disables breakpoints at specific statements.

Syntax

```
BREAK ON N1,...Nn

or

BREAK OFF  N1,...Nn
```

where N1...Nn is a list of statement numbers separated by commas.

If a statement at which a breakpoint is set is accessed during the execution of a program, control is returned to the BASIC processor command level (immediate mode) before the statement is executed.

If no statement numbers are specified with a BREAK OFF statement, all breakpoints previously set ON are set OFF.

Example:

```
90  BREAK ON  40, 318, 215, 10, 45, 9999

...

195 BREAK OFF 10, 40

200 FOR X = 1 to 10

210 A = FNA (X)

215 REM, CHECKING VALUE OF A

220 NEXT X

...

235 BREAK OFF 215
```

CALL

The CALL statement is used to interface to a written subroutine that is user-written in FORTRAN or assembly language.

Syntax

```
CALL C

or

CALL  C(L1, L2....Ln)
```

where the constant C is an integer that serves as a subroutine identifier. The value of the constant C is limited only by the size of available memory; i.e., as many subroutines as will fit in memory may be called. The subroutine identifier is related to the address of the subroutine by a user supplied file. The format and use of this file are described in Section 7, Interface Conventions.

L1...Ln are items in a list that are argument specifications to the subroutine calling sequence. The argument list may contain up to 26 items. An argument specification can be a numeric or string variable, a numeric expression, an array, a subscripted variable or a function argument. String expressions or string constants cannot be included in the argument list. Arrays, variables, or subscripted variables can be redefined by the called subroutine. However, the value of numeric constants or expressions cannot be redefined by the called subroutine; they can only be passed to the called subroutine. All items in the list L1...Ln must be separated by commas.

Example

```
CALL  5 (X1, X2, 6, A(10), X+1)
```

DATA

The DATA statement allows the user to specify a list of numeric or string constants within the program. The constants must be accessed by a READ statement.

Syntax:

```
DATA  C1, C2, C3,...,Cn
```

where C1...Cn are numeric and/or string constants separated by commas. A trailing comma causes an error. The list of string or numeric constants may be any length as long as the length of the line is not exceeded. To extend the list of constants more than one line, it is permissible to write subsequent DATA statements.

The DATA statement is a nonexecutable statement that creates a block of data to be read by the READ statement. BASIC separates numeric constants in DATA statements from string constants and maintans a separate data pool for each type. Any number of DATA statements can appear at any place in the program. Data from all of the DATA statements in the program, taken in the order of the DATA statements, are concatenated to create a block of numeric DATA and/or a block of string data.

When there are no more DATA items to be read, the program prints the message:

```
'OUT OF DATA AT N'
```

where N is a statement number; and the program terminates.

Examples:

```
100  DATA  2.3, 3.4, 3.7E02, 1, 2, 3

200  DATA  3.1415, 2.783, 0

300  DATA  'ITEMS', 300  'COST'  1.58
```

DEF

The DEF statement defines a function of a single variable.

Syntax

```
DEF  FNA(V)
```

where A is the function name and V is any variable. V may be an expression that returns a value. For further explanation, refer to "User Defined Functions" in Section 3.

The DEF statement defines a single-line function whose value is the value of any expression that can refer to the optional function parameters. The type of the expression must be the same as the type of the function as defined. A particular function cannot be defined by more than one DEF statement in the same program.

A function parameter (function term) is a scalar variable that is local to the function body, and a function parameter has no relationship to a variable of the same name elsewhere in the program. The value of the function parameter is set to the value of the corresponding function argument when the function is invoked.

DEF is a non-executable statement, and a DEF statement can be written anywhere in the program.

Examples:

```
 20  DEF  FNX (B)  =  2./COS(B)*3

100  DEF  FNO (P) = 3.14159
```

DEFINE FILE/DEFINE READ FILE

The DEFINE FILE statement opens the specified BASIC logical file unit for reading and writing.

Syntax:

```
DEFINE FILE #E1 = 'S', M, E2
```

where E is an arithmetic expression defining file unit numbers (1-8), S is a string expression specifying file names or an I/O device, M is an optional parameter that specifies the mode of the file, and E2 is an optional parameter that defines file record size.

```
DEFINE READ FILE #E1 = 'S', M, E2
```

The DEFINE READ FILE statement functions the same as DEFINE FILE, except it opens the specified BASIC Logical Unit for reading only. The parameters have the same meaning as in the DEFINE FILE statement. For further examples of usage of DEFINE FILE and DEFINE READ FILE, refer to Appendix C.

E1 is an arithmetic expression defining BASIC logical unit number. BASIC allows eight logical units (1-8).

Device Names

S is a string expression defining file name. If the name starts with a left parentheses, it is interpreted as a device name of the format: (dxu)

where: d = device identifier
x = don't care
u = unit specifier

Possible values for device identifier are:

A - ASR - pdev = 1
P - PTR/P - pdev = 2
C - Cards - pdev = 3
L - Line Printer - pdev = 4
M - Magnetic Tape - pdev = 5

The unit specifier, u, ranges from 0 to 9. If the unit specifier is not a digit, the physical unit is the BASIC unit plus 3.

Disk File names are one to six characters long and begin with a letter. I/O device identifiers are enclosed in parentheses and delimited by single quotes. Valid I/O device identifiers are as follows:

Device Identifier	Device
'(A)'	Teletype (terminal)
'(P)'	Paper tape reader/punch
'(LPR)'	Line printer
'(C)'	Card reader
'(MT1)'	Magnetic Tape #1
'(MT2)'	Magnetic Tape #2
'(MT3)'	Magnetic Tape #3
'(MT4)'	Magnetic Tape #4

The standard versions of BASIC do not contain drivers for physical devices 3 to 5. They can be configured by modifying the BASIC IOCS configuration module, BASIO, and rerunning the appropriate command file

File Modes

The optional mode parameter M specifies the mode of the file (i.e., the kind of file that it is). Possible entries are:

Mode (M)	Meaning
ASC	ASCII file.
ASC SEP	ASCII file. When writing to the file, BASIC inserts a comma between output fields rather than the spaces specified by the WRITE statement item separators. The type of file produced by specifying ASC SEP is suitable for input to other BASIC programs (i.e., acceptable to BASIC as a READ file).
BIN	Binary file. Data written into this type of file is in internal memory format instead of being converted to ASCII strings. An arithmetic item generates two words of data in the file, a string item generates (C+2)/2 words of data (where: C is the number of characters in the string).

Mode (M)	Meaning
BIN DA	Same as BIN mode, except: 1. Fixed length records are written. 2. The file is opened as a DAM file (Refer to the Disk and Virtual Memory Systems User Guide). 3. The POSITION statement operates on the file.

If the mode parameter, M, is omitted; its value is considered to be ASC.

Record Size

The optional parameter E2 is an arithmetic expression that defines the record size of a file (number of words/record). The value of E2 may range from 2 to 512. If the field E2 is omitted, a value of 60 is assumed. The parameter E2 must be specified if mode M is specified as BIN DA.

Examples of Use of DEFINE FILE:

```
10  REM CARD TO PRINT CONVERSION,  DECK  1000 CARDS

20  DEFINE FILE #1 = '(CRD)'

30  DEFINE FILE #2 = '(LPR)'

35  REM N=CARD NUMBER; N$=BLANK OR END OF DECK;
                       C$=CARD IMAGE

50  FOR I = 1 to 10000

60  READ FILE #1, N, C$, N$

70  IF N$ = 'END OF DECK' THEN 99

75  REM STATEMENT 70 SHOWS ONE WAY TO HANDLE END OF FILE

76  REM SITUATIONS

77  REM SEE 'ON END' ...

80  WRITE FILE #2, C$

90  NEXT I

99  END
```

DIM

The DIM statement defines the number and size of the dimensions of a numeric array or string array.

Syntax:

```
DIM A(C1)

or

DIM A(C1, C2)
```

where A is a numeric or string array name and C1, C2 are unsigned numeric constants that specify the upper bounds of the corresponding dimension.

The DIM statement specifically defines array names, establishes the number of dimensions (one or two), and specifies the number of elements in each dimension. The lower bound of each array dimension is always 0. The upper bound of each array dimension is that value specified for the element in the DIM statement. (C1 + 1) locations are allocated for a single-dimension array (vector); and ((C1 + 1) * (C2 + 1)) locations are allocated for a two-dimension array.

Any number of DIM statements can appear in a program. However, an array name can be explicitly defined by a DIM statement only once in a program. (However, it can be redimensioned any number of times by subsequent MAT statements). A DIM statement is nonexecutable.

Examples:

```
100  DIM A(12)
```

declares a one-dimensional numeric array of thirteen locations (A(0)...A(1])).

```
300  DIM A$(2,3)
```

declares a two-dimensional string array of 3 columns (0, 1, 2) and 4 rows (0, 1, 2, 3).

NOTE: The arrays defined by DIM statements may be used later as matrices (e.g., the set of array dimensions that are non-zero). These operations are discussed in Section 6.

END

The END statement terminates execution of the BASIC program.

Syntax

END

The END statement indicates the end of the main program. It is equivalent to and has the same function as the STOP statement.

Examples:

```
9999  END
```

When this statement is executed, the message:

```
END AT 9999
```

is printed.

FOR

The FOR statement defines the beginning of a loop, (sequence of statements to be executed more than once within the program). The NEXT statement must be used subsequent to the FOR statement to define the end of the loop.

Syntax

```
FOR V = E1 TO E2

or

FOR V = E1, E2

or

FOR V = E1 TO E2 STEP E3

or

FOR V = E1 TO E2, E3

or

FOR V = E1, E2, E3
```

where V is a scalar numeric variable; and E1, E2, and E3 are numeric expressions. The variable V is the control variable of the loop. The first expression (E1) defines the initial value of V. The second expression (E2) defines the final value of V. The expression E3 is optional and is the incremental value added to V when the subsequent NEXT statement is executed. The words TO and STEP may be omitted and replaced by commas.

When the "STEP E3" or "E3" term is omitted, the value +1 is used.

The value of the control variable (V) can be modified within the loop. Its value will be available at the end of the loop. Also, the loop may contain statements that jump out of the loop.

FOR-NEXT loops can be nested indefinitely as long as available memory is not exhausted. FOR-NEXT loops cannot be interleaved. A nested FOR-NEXT loop cannot use the same control variable as the FOR-NEXT loop that contains it.

```
  5  REM ANOTHER EXAMPLE
 10  PRINT 'PLEASE SPECIFY N; '
 12  INPUT N
 14  PRINT 'PLEASE SPECIFY M;'
 16  INPUT M
 20  DIM B(1000)
 30  FOR I=N TO M STEP .1
 40  LET B(I)=3.1416*I↑2
 50  PRINT I,B(I)
 60  NEXT I
 66  STOP


RUN
PLEASE SPECIFY N;
2
PLEASE SPECIFY M;
3.5
2              12.5664
2.1            13.8545
2.2            15.2053
2.3            16.6191
2.4            18.0956
2.5            19.635
2.6            21.2372
2.7            22.9022
2.8            24.6301
2.9            26.4208
3              28.2744
3.1            30.1907
3.2            32.17
3.3            34.212
3.4            36.3169
3.5            38.4846

STOPPED AT LINE 66
```

The next example of FOR-NEXT assigns values to the elements of a single dimension array.

```
110  DIM  X(10)

110  FOR I = 0 TO 10

120  READ X(I)

140  NEXT I

300  DATA  0,1,2,3,4,5,6,7,8,9
```

One of the common reasons for using FOR-NEXT loops is to deal with two-dimensional arrays. The idea is to use two subscript

variables to point to the column and row of the array controlled by a loop. This is illustrated in the following example:

```
100  READ C1,C2
110  FOR I=1 TO C1
120  FOR J=1 TO C2
130  LET A(I,J)=0
140  NEXT J
145  NEXT I
148  REM ELEMENTS OF ARRAY ASSIGNED TO ZERO
150  READ C3
160  IF C3=0 THEN 300
170  READ C4,X
180  LET A(C3,C4)=X
190  GOTO 150
200  REM STATEMENTS 150 TO 190 ASSIGN VALUES FROM THE DATA LIST TO
202  REM ELEMENTS OF ARRAY A.
300  FOR I=1 TO C1
305  FOR J=1 TO C2
310  PRINT A(I,J)
320  NEXT J
325  PRINT
330  NEXT I
350  REM ABOVE LOOP PRINTS VALUES OF ARRAY ELEMENTS.
400  DATA 3,4
405  DATA 1,1,16,1,2,256,1,3,512,1,4,1046
410  DATA 2,1,34,2,2,300,2,3,13,2,4,9.87654E+08
420  DATA 3,1,99,3,2,88,3,3,7777,3,4,56
440  DATA 0
999  END

RUN
 16
 256                    Values assigned to A(1,1),..., A(1,4)
 512
 1046

 34
 300
 13                     Values assigned to A(2,1),..., A(2,4)
 9.87654E+08

 99
 88
 7777                   Values assigned to A(3,1),...,A(3,4)
 56

END AT LINE 999
```

GOSUB

The GOSUB statement allows control to be passed to an internal subroutine.

Syntax

```
GOSUB  N
```

where N is a statement number in the program, N is a statement number that specifies the line at which the internal subroutine is to start. The subroutine must contain a RETURN statement.

The GOSUB statement saves the line number of the statement that follows it, and then transfers to the statement specified by the line number N. When a RETURN statement is subsequently executed, control returns to the statement whose line number was saved (i.e. the statement that follows the referencing GOSUB statement).

A subroutine may itself contain a GOSUB statement. Up to eight GOSUB statements may occur before the execution of a RETURN statement. RETURN always causes control to be returned to the statement following the most recent outstanding GOSUB statement.

Examples:

```
173  GOSUB  1000
```

The following is an example of a trivial but valid program; the statements are executed in the order: 10, 30, 50, 70, 60, 40, 20.

```
10  GOTO  30

20  STOP

30  GOSUB  50

40  RETURN

50  GOSUB  70

60  RETURN

70  RETURN
```

GOTO

The GOTO statement causes program control to be passed to a non-local, designated statement.

Syntax

```
GOTO  N
```

where N is a statement number of a valid statement.

The GOTO statement causes program execution to continue at the statement specified by N.

Examples:

```
 10  GOTO  75

200  GOTO 400
```

Example use of GOTO:

```
100  PRINT 'INITIAL VALUE'

110  INPUT I

120  PRINT 'TYPE CHANGE'

130  INPUT C

135  REM C IS + OR -

138  IF C = 0  THEN 999

140  LET I = I + C

150  PRINT 'NEW VALUE IS', I

160  PRINT

180  GO TO 120

999  END
```

IF

The IF statement allows processing to be dependent on the true or false value of a relational expression.

Syntax

```
IF E1 rel E2  THEN  N

or

IF E1 rel E2 GO TO N
```

where E1 and E2 are either both numeric expressions or both string expressions; rel is one of the following relational operators:

Operator	Meaning
<	less than
>	greater than
<= or =<	less than or equal
>= or =>	greater than or equal
=	equal
<> or ><	not equal

N is either a statement number or a statement, including another IF statement. (N can only be a statement if the verb is THEN.)

If E1 and E2 satisfy the relation specified by rel, control is transferred to the statement specified by N; otherwise, execution continues with the statement that follows the IF statement.

Examples

```
100  IF  A$ = 'YES'  THEN 125

200  IF  ABS (X-Y) < E1  THEN 75

205  IF  C1 => C2  GOTO  50

305  IF  X <>0 THEN IF X < 100 GOTO 402

402  IF  (TAN(X9)-1) = (T(J*2-1) ↑ 3  THEN 350
```

If any of the above conditions are false, program execution continues with the statement that follows the IF statement.

INPUT

The INPUT statement requests data from the user terminal.

Syntax

INPUT L1, L2,...Ln

where L1,...Ln is a list of references separated by commas. Trailing commas are ignored. If more items are input then are on the specified list L1...Ln, the additional items are discarded.

The INPUT statement causes data to be read from the users terminal and assigned to the references in the list L1...Ln in the order that they are typed. If there are any array references in the list L1...Ln, subscript expressions are not evaluated until all references that precede the subscript expressions in the input list have been assigned values.

The INPUT statement prints the prompt-character, !, to indicate that input is desired. The user must be sure to type input as his program requires.

Data items provided by the user must match the data type of the corresponding reference in the list, L1...Ln, in the INPUT statement.

A single quote may be combined in a string typed in response to an INPUT statement. It is transferred literally to the program area. Example, typing:

ABC'D

in response to an INPUT statement puts the string, ABC'D, in the program storage area.

When a numeric value is expected, all characters up to the next comma or CARRIAGE RETURN are input to the program. Spaces, blanks and tabs are ignored.

Examples

```
10  INPUT  I1
20  FOR  I2 = I1,  10
30  INPUT  A (I2)
40  NEXT  I2
```

Sample Output

```
    RUN
!6
!12345
!1.3141579
!2.45
!9999999999
!34
```

In the above example, the ! characters are typed by the system; the numbers are input by the user in response to them.

Interrupting INPUT

The user can stop typing in a series of values in response to an INPUT statement in his program and return to BASIC command level by typing CONTROL-C (pushing the control and C key simultaneously. Example:

```
>10 INPUT A, B, C, D, E
>20 PRINT A, B, C, D, E
>RUN
! 1, 311 CTL-C          ←User interrupts INPUT
END OF DATA AT LINE 10 ←Response from BASIC
>                       ←Return has been made to
                         command level
```

LET

The LET statement allows an arithmetic variable or string variable to be assigned a value.

Syntax

```
LET  V  =  E

or

V  =  E
```

where V is a numeric variable or a string variable, and E is an expression of the same data type as V.

The LET statement assigns the value of an expression to one or more scalar variables or subscripted array elements. Subscripts in the expression E are calculated before the expression is evaluated and before any assignment is done.

Scalar arithmetic variables not explicitly assigned a value are assigned a default value 0 when first referenced in a program. Unassigned scalar string variables are assigned a value of a null string ('').

Array elements not explicitly assigned a value are given a default assigned value when the array is referenced. (See Section 2.)

Examples

```
 10  I  =  20

 20  LET  I  =  2

100  LET  X(5)    =  24

102  LET  V  =  C

110  LET  A$  =  'STRING OF CHARACTERS'

120  LET  A$  =  B$ + C$

440  LET  I3  =  5

500  A(J)  =  SIN(X-4.5) + Q3

500  LET  S$(J+5)  =  M$  + '.00'
```

NEXT

NEXT is used in conjunction with the FOR statement to increment the control variable of the FOR-NEXT loop.

Syntax

```
NEXT  V
```

where V is the control variable used with the previous FOR statement.

Refer to the description of the FOR statement for further details.

The NEXT statement marks the end of a FOR-NEXT loop; it is always used in conjunction with a preceding FOR statement.

Example

```
700  FOR  I  =  1 TO 100

705  LET  A  =  A + 1

713  NEXT  I
```

ON

ON allows control to be passed to one of a list of statements depending on the value of an expression.

Syntax

```
ON  E  GOTO  N1, N2, N3...Nn
```

where E is an expression and N1...Nn are numeric expressions separated by commas that represent statement numbers.

The ON statement uses the value of the numeric expression to select one of the statement numbers as the target of a GOTO operation. The value of the expression is truncated to yield an integer that must be positive and also must be less than or equal to the number of statement numbers (Nn) specified in the ON statement.

Examples:

```
...

20  ON  (I-1)  GOTO 100, 200, 300, 400
```

If I = 1, control goes to statement 100; if I = 2, control goes to 200; if I = 3, control goes to 300; and if I = 4, control goes to 400.

The ON statement is useful because the IF statement provides only a two-way branch in a program. The ON statement can provide more alternatives (i.e., a multi-way branch).

ON END

The ON END statement directs the transfer of control to a given statement when an End of File is reached during a READ or POSITION operation on the unit specified in the ON END statement.

Syntax

```
ON END #E  GOTO N
```

where E is an expression that specifies a BASIC logical unit (1-8) (Refer to DEFINE FILE); and N is a statement number. The ON END statement does not test for End of File; it establishes action to be taken when the last file record is read.

Example:

```
10  DEFINE FILE #1 = 'INPUT'
...
40  ON END #1 GO TO 20
50  READ #1, A$, A, B$, B
...
```

POSITION

POSITION positions a file on the unit specified to the start of the record specified.

Syntax

```
POSITION  #E1 TO E2
```

where E1 is an expression that specifies the BASIC logical unit (1-8) and E2 is an expression that specifies the record in the file. Record numbering starts at one. The unit (E1) must have been defined to be BIN DA mode (refer to DEFINE FILE).

If the record number specified is greater than the number of records in the file, the file is positioned to the End of File and the ON END action is taken.

PRINT

The PRINT statement causes information to be printed at the terminal.

Syntax

PRINT L1, L2,...,Ln

where L1...Ln are 0 or more items in a list separated by commas or colons. Individual list items L1...Ln may be either numeric expressions or string expressions.

The PRINT statement generates lines of output to be printed at the terminal. A single PRINT statement can generate either one line, several lines, or partial lines of information.

The format of the terminal line image is determined by the elements in the print list. Each element in the list L1...Ln is evaluated to yield a string of characters to be placed on the terminal print line.

Printing Numeric Expressions

A PRINT list item that is a numeric expression is evaluated and converted to the equivalent character string representation. This string begins with the sign character and ends with a blank.

If the value of the expression is positive, a blank is printed for the sign character. If the value of the expression is negative, a minus sign is printed for the sign character.

Integers: Numbers printed as integers consist of a string from one to six decimal digits without a decimal point. Examples:

```
 14

-20796

 1
```

Fractions: Numbers up to six decimal digits may be printed with a decimal point.

Fractional format is used for nonintegers with an absolute image in the range .1 to 99999.5. Examples:

```
2.5

12.4 3

-0.00796

0.00371

7.74186
```

Scientific Format: A number printed in scientific format is of the form:

```
X E + Y

or

X E - Y
```

where X is a fractional number greater than one and less than ten, and Y is an integer power of 10 ranging from -38 to +38. Scientific format is used whenever integer or fractional format cannot be used as shown in the following example:

```
LET X = 999999

LET X = X+1

PRINT X
```

results are printed:

```
1.0 E+6
```

Other examples of numbers in scientific format are:

```
2.54 E+13

5.0 E+5

-1. E-32
```

Printing String Expressions

A string expression in the PRINT list L1...Ln is evaluated and the resulting string of characters is printed in the output at the teletype. BASIC does not interpret contents of this character string; therefore, unpredictable results may occur from the inclusion of characters that do not advance the print line by one position (such as combinations of a backspace with other characters).

Comma Separator

The output from the PRINT statement is normally divided into zones of 14 characters each. The first zone starts in column 0, the second in column 14, etc. The number of zones is determined by characters, five zones are printed.

A comma in a print list causes the Teletype to advance to the first character position of the next available zone. If character overflow occurs, the current line is printed and a new line is started. If the last element of the print list is a comma, the partial line, if any, is printed; and the Teletype is positioned at the start of the next available zone.

Example Use of Comma in PRINT Statement

The statement:

```
100  PRINT  I, J, K, L
```

might result in the following output:

```
1.0          2.4          1.416          75
```

Colon Separator

A colon in a PRINT list is used to separate PRINT elements and inhibits the printing of items in different zones. A colon specifies that the preceding items to be printed is to be followed by a space rather than the number of spaces required to position to the next print field.

Examples of Use of Colon in PRINT Statement

The previous example written as follows:

```
100  PRINT  I: K
```

causes the following output:

```
1.024  1.41675
```

The statement:

```
200  PRINT  'A': 'B', 'CAT':  'DOG'
```

prints:

```
A  B              CAT DOG
```

Tab Request

The tab print element requests that the Teletype be moved to a specific character position (column). The tab request is written as:

```
TAB (E)
```

where E is a numeric expression. An example of the tab request is:

```
100 PRINT X: TAB(40): Y
```

PRINT List Termination

If the print list does not end in a comma or colon, a CARRIAGE RETURN character is appended to the print output and the line is transmitted to the terminal. A null (empty) PRINT list causes the previous line to be finished or a blank line to be printed.

PRINT Statement Examples

```
20 PRINT X, SIN (Z 2 - Y 2)
30 PRINT 'VALUE IS': X-Y
40 PRINT ' ', A$ + SUB (B$, I, J)
50 PRINT
```

Example of Use of Print for Conservational Input/Output

```
10 PRINT 'ENTER LENGTH IN INCHES':
20 INPUT L$(1,1)
30 LET X4 = L(1,1)/12
40 PRINT X4: 'FEET'
```

Sample results:

```
ENTER LENGTH IN INCHES ! 30
 2.50  FEET
```

PRINT USING

A formatted print-statement (the PRINT USING statement) generates formatted output.

Syntax

```
PRINT USING S$, L1, L2...Ln

or

PRINT USING  S$, L1: L2...Ln
```

where S$ is a string expression and L1...Ln are items in a list that are string or numeric expressions specifying values to be printed, separated by commas or colons.

A single PRINT USING statement can generate one line, several lines, or a partial line of printed output. The characters generated by a PRINT USING statement are formatted as specified by a control string.

Format Fields

The string specified by S$ contains a description of the editing to be applied to the values in the list L1...Ln. The string S$ is divided into a series of fields each of which controls the formatting of a single value in the PRINT list L1...Ln. The fields describe a numeric or string value.

There are seven special characters for defining numeric fields in the format. These characters are:

```
# . , ↑ + - $
```

Their use in a format field is described in the following tables and paragraphs.

There are three special characters for defining string fields in the format. These are:

```
< > #
```

Their use in a format field is described under the heading "String Fields".

Numeric Fields

Pound Sign (#): For each pound sign in the field descriptor, a digit (0-9) from the output value is substituted. Examples are shown in the following table.

Field Format	Datum	Representation	Remarks
#####	25	25	Right justify digits in field with leading blanks.
#####	.-30	30	Signs and other non-digits are ignored.
#####	1.95	2	Only integers are represented; the number is rounded to an integer.
#####	598745	*****	If the datum is too large for the field, all asterisks are printed.

Table 5-2. Pound Sign in Descriptor Field

Decimal Point (.): The decimal point places a decimal point within the string of digits in the fixed character position in which it appears. Digit positions to the right of the decimal point are not blank filled. Examples are shown in the following table.

Field Format	Datum	Representation	Remarks
#####.##	20	20.00	Fractional positions are filled with zeroes.
#####.##	29.347 0.079	29.35 0.08	Rounding occurs on fractions.
#####.##	789012.34	******.**	When the datum is too large, a field of all asterisks, including the decimal position, is printed.

Table 5-3. Decimal Point in Descriptor Field

Comma (,): A comma in a descriptor places a comma in the output record at that character position unless all digits prior to the comma are zero, in that case, a space is printed in that character position. The following table gives examples of use of the comma.

Field Format	Datum	Representation	Remarks
+$,###.##	30.6	+$ 30.60	Space printed for comma when leading digits blank.
+$#,###.##	2000	+$2,000.00	Comma printed.
++##,###	00033	+00,033	Comma is printed when leading zeroes are not suppressed.

Table 5-4. Comma in Descriptor Field

Vertical Arrow (↑): A string of four vertical arrows can be used to indicate an exponent field which is filled by E±n where n is a two digit integer. The following table gives examples of use of the vertical arrow.

Field Format	Datum	Representation
+##.## ↑↑↑↑	170.35	+17.04E+01
+##.## ↑↑↑↑	1.2	-20.00E-02
++##.## ↑↑↑↑	6002.35	+600.24E+01

Table 5-5. Vertical Arrow in Descriptor Field

Plus or Minus Signs (+ -): A single plus sign as either the first or last character in the format descriptor causes a + to be output if the data item is positive, or a - if the data item is negative.

Two or more plus signs starting at the first character of the descriptor cause the sign to be output (+ if positive, - if negative) immediately to the left of the most significant nonzero digit of the output item. If required, the second through the last plus sign is used as digit positions as required by the magnitude of the number.

A minus sign (or signs) has the same effect as plus signs , except a space is output for a positive sign. The following table gives examples of the use of + or - in formatted print output.

Field Format	Datum	Representation	Remarks
+##.##	20.5	+20.50	
+##.##	1.01	+ 1.01	Blanks precede the number.
+##.##	-1.236	- 1.24	
+##.##	-234.0	******	When the datum is too large for the specified format a field of all asterisks is printed.
###.##-	20.5	20.50	
###.##-	000.01	0.01	The last leading zero before the decimal point is not suppressed.
###.##-	-1.236	1.24-	
###.##-	-234.0	234.00-	
---.##-	-20	-20.00	Second and third signs are treated as digit positions (#) on output.
---.##	-200	******	When the datum does not agree with the specified field, asterisks are printed.
---.##	2	2.00	

Table 5-6 Plus and Minus in Descriptor Fields

Dollar Sign ($): A single dollar sign as either the first or second character in the descriptor causes a dollar sign to be output in that position of the output record.

Multiple dollar signs starting at either the first or second character of the descriptor cause a dollar sign to be placed immediately to the left of the most significant nonzero digit. The only character that may precede a dollar sign in a format descriptor is a fixed sign (+ or -). The following table gives examples of use of the $ in formatted print output.

Field Format	Datum	Representation	Remarks
-$###.##	30.512	$ 30.51	
$###.##+	-30.512	$ 30.51-	
+$$$$#.##	13.20	+ $13.20	Extra $ signs may be replaced by digits as with floating + and - signs.
$$##.##-	-1.0	$01.00-	Leading zeroes are not suppressed in the # part of the field.

Table 5-7. Dollar Sign in Descriptor Field

String Fields

Pound Sign (#): Each pound sign in the descriptor field represents a character position from the second to the nth character position. A character from the output (i.e., letter, numeral, or symbol) is substituted in that position.

Examples are shown in Table 5-8.

Left Angle Bracket (<): This character in a descriptor field is always positioned first when it is used. It represents the first character position and the first character from the output is substituted for it. It also designates that the output string is to be left justified in the PRINT statement field. An example is shown in Table 5-8

Right Angle Bracket (>): This character in a descriptor field is always the first character of the field. The first character of the output is substituted for it and it designates that the output string is to be right justified in the PRINT field. Table 5-8 shows an example.

Field Format	Datum	Representation	Remarks
> ######	TWELVE	TWELVE	right-justified
< ######	TWELVE	TWELVE	left-justified

Table 5-8. String Descriptor Fields

Print Using Statement Example

```
150 REM *****EXAMPLE TO SHOW VARIOUS USES OF PRINT USING
160 REM
170 INPUT A,B,C
180 LET E$="EX--1"
190 PRINTUSING "<####################",E$
200 PRINTUSING ">####################",E$
210 REM LAST TWO LINES SHOW HOW JUSTIFICATION WORKS
220 LET F$="-##.##"
230 PRINTUSING F$,A,B,C
240 PRINTUSING "$$######.##",A,B,C
250 PRINTUSING ">########## TO 2",E$
260 REM NOTE CONCATENATION, RESULT IS PLACED IN FIELD SPECIFIED
270 INPUT X
280 PRINTUSING "-##.##",SQR(X)
```

Sample Output

```
>RUN
?123.34,3345.93,45
EX--1
                    EX--1
 23.34
******
 45.00
 $00023.34
 $03345.93
 $00045.00
      EX--1 TO 2
?654
 25.57

>
```

Printing Special Characters

To print a literal copy of one of the characters used with special meaning in a format field, a string field must be used with the PRINT statement and the character must be passed as part of the print list. For example, the following statement prints a period at the end of the output line.

```
10  PRINT USING 'X IS -### ':'.',X
```

If the statement were written

```
10  PRINT USING 'X IS -###.',X
```

the decimal point would be part of the numeric field output.

READ

The READ statement is used in conjunction with a DATA statement. DATA defines a series of data values (literals); READ sets a list of variables equal to literals in the numeric and/or string data pools.

Syntax

```
READ L1, ..., Ln
```

where L1, ..., Ln is a list of references, which may be numeric variables, string variables or arrays, separated by commas.

The READ statement causes numeric or string values stored in the data pools by DATA statements to be assigned, starting at the next available element in the applicable data pool. The assignments are made in the order specified by the references in the list specified with the READ statement.

Subscript expressions in an array reference in the list L1, ..., Ln are not evaluated until all preceding references have been assigned values.

If a data list is exhausted, a message is printed and program execution is halted.

The RESTORE statement may be used to prepare to read the data again.

Examples:

```
100  READ  X, Y, Z

110  READ  X$, X, Y$, Y, Z$, Z

120  READ  X(3)
```

For examples of READ, all of the DATA are treated as a single list of numbers. Each READ operation takes the next available number from the list and advances one position on the list. The following example illustrates this principle:

```
10  DATA  1.314  1.817
20  DATA  1, 2, 3, 5, 8, 13, 21, 34
30  DATA  55, 89
40  READ  N1
50  READ  N2
60  FOR K1+N1, N2
70  READ  A(K1)
80  NEXT  K1
90  RESTORE
```

READ FILE

Input may be read from a formatted file prepared by the system editor, from a file created by another BASIC program or from a binary file created by a FORTRAN program. The format of the files and their types and modes is defined by the DEFINE FILE statement.

Syntax

```
READ #N, L1, ..., Ln
```

where N is a file number and L1, ..., Ln are a list of all numeric variables or all string variables separated by commas.

This variation of the READ statement reads from the file specified by #N.

Initially, the READ FILE statement forces the reading of a new record. The READ FILE statement reads values from the file starting with the first data item in the record currently pointed to and the file pointer is incremented by 1 after each data value is read.

If a file number specified in a READ FILE statement has not been defined in a previous DEFINE FILE statement, the message:

```
ERROR UF AT LINE N
```

(where N is a statement number) is printed and execution of the BASIC program halts, and the user's program returns to BASIC command level.

Examples:

```
100  READ  #4, V(I), A
110  READ  #4, A1, A2, A3
```

READ * FILE

Syntax

READ * #N, L1, ... Ln

The READ * FILE statement has the same effect as the READ FILE statement except it does not initially force a new record to be read from the unit specified. If data remains in the last record read from the unit, it is used before the new records are read.

REM

This statement identifies a remark. It is not executed.

Syntax

REM S

where S is any string of ASCII characters not including the carriage return character.

The string of characters following REM is ignored by the BASIC interpreter. The REM statement has no effect on the program; it is provided for the convenience of the user.

Example:

```
10  REM PROGRAM TO PERFORM MEDIA CONVERSION
20  REM MLG MODIFIED BY SDH 10-15-72
30  REM
40  REM
```

RESTORE

The RESTORE statement resets the DATA list pointer so that the list may be re-used by subsequent READ statements in the program.

Syntax

```
RESTORE
RESTORE #
RESTORE $
```

The RESTORE statement re-initializes either or both of the data pools. The next read statement executed reads the first data item in the pool or pools restored.

The RESTORE statement resets each data pool. The RESTORE $ statement resets the string data pool only. The RESTORE # statement resets the arithmetic data pool only.

Example

```
112  READ  A, B
115  LET  C= A*B
120  PRINT  A: '@' B, "PRICE":  C
130  RESTORE
135  READ  Z
140  PRINT  'NO OF ITEMS IS':Z
900  DATA  100, 3.50
```

Output is:

```
100  @  3.50      PRICE  350.
NO OF ITEMS IS 100
```

RETURN

The RETURN statement causes control to be returned from the subroutine that contains it to the statement immediately following the GOSUB statement that invoked the subroutine (i.e., the last outstanding GOSUB).

Syntax

```
RETURN
```

Examples:

```
100  INPUT  A
110  GOSUB  300
111  INPUT  A$
120  IF  A$  <>  'END'  THEN 100
130  END

300  REM  'SUBROUTINE TO CALCULATE IF A'
301  REM  'NUMBER N IS PRIME'
310  FOR  X = (A-2) TO 1 STEP -1
320  LET  Q1 = A/X
325  LET  Q2 - INT(X.X)
330  LET R = Q1 - Q2
340  GOSUB  400
350  NEXT  X
360  IF R = 0  THEN 380
370  PRINT 'NUMBER'  :A:  'IS A PRIME'
380  RETURN
400  IF  R ><  THEN  420
410  PRINT 'NUMBER'  :A:  'IS NOT A PRIME'
420  RETURN
```

The RETURN statement in statement number 380 causes a return to the statement 111; the RETURN statement in 420 causes a return to 350.

REWIND

The REWIND statement causes the specified I/O unit to "rewind".

Syntax

```
REWIND  #N
```

where N is an arithmetic expression defining a file unit (1-8).

If the REWIND statement refers to a disk file, it is reset to start from the first record.

Examples:

```
...
100  DEFINE FILE #4 = 'ALPHA'
110  INPUT N
120  FOR  I = I TO N
130  READ  #4, A
140  NEXT  I
150  REWIND #4
```

STOP

STOP causes the program to return to its caller.

Syntax

```
STOP
```

Any files opened by the program are closed. Executing a STOP statement in a program is equivalent to an END statement.

Example

```
9999  STOP
```

causes a message to be printed such as:

```
STOPPED AT 9999
```

TRACE

The TRACE statement is used to turn trace mode ON or OFF.

Syntax

```
TRACE  ON
```

or

```
TRACE  OFF
```

When trace mode is ON, the statement number of each statement is printed prior to its execution.

TRACE is useful in debugging a program that contains many GOTO and/or GOSUB statements.

Examples:

```
110  TRACE  ON
115  FOR  I  =  1 TO 10
120  A3 = A1 + FNX (I) -3.1
130  IF  A3  < 0  THEN 400
150  GOSUB 6000
160  IF  A3 = 0  THEN 500
170  GOSUB 7500
180  IF  A3 > 0  THEN 600
190  GOSUB  9000
195  NEXT  I
200  TRACE OFF
```

Assuming all conditions are true (in the first pass) a partial view of the trace might look as follows:

```
[115]
[120]
[400]
[401]
...
[499]
[150]
[6000]
...
[6099]
[170]
[7500]
[7502]
...
[7550]
[180]
[600]
...
[650]
[190]
[9000]
[9010]
[9020]
[195]
[115]
...
```

WRITE FILE

The WRITE FILE statement directs output to a file.

Syntax

```
WRITE  #N

or

WRITE  #N, L1,...Ln
```

where N is an expression that yields a file number (1-8) and L1,...Ln is an optional list of all numeric variables or all string variables separated by commas or colons.

A print element in the list can be an expression or a TAB request.

WRITE statement output lines are appended to the specified file in a stream.

Either full lines (terminated by a CARRIAGE RETURN character) or partial lines (terminated by a comma or colon) may be output to a file.

Read After Write Check

If an attempt is made to read on a unit after a WRITE has been performed, without an intervening REWIND or redefinition of the unit, a WR error diagnostic is printed. This check does not apply in the case of writing BIN DA files.

Example

```
10  DEFINE FILE  #1 = '(LPR)'
20  FOR  I  =  1 TO 100
30  WRITE FILE  #1,'ITEM-':X, 'COST-$ ' Y, 'ONE EACH'
40  NEXT  L
120 DEFINE FILE #2 = 'ALPHA'
130 FOR  X  =  1  TO  100
135 LET  N  =  X2
140 WRITE  #2, X, N
150 NEXT X
```

Statements 10 to 40 print 100 lines on the line printer (if it is assigned); statements 120 to 150 consecutively write 100 values of X and 100 values of N onto a disk file ALPHA.

WRITE USING

Formatted output strings may be passed to a file by means of the WRITE USING statement.

Syntax

```
WRITE USING S$, #N, L1,...,Ln
```

where N is a file number (1-8); S$ is a string expression, as in the WRITE USING statement; and L1,...,Ln are a list of expressions separated by commas or colons.

This variation of the WRITE USING statement directs output to be appended to a Teletype formatted file. A single WRITE USING statement can generate one line, several lines, or a partial line of output.

Example:

```
140  WRITE USING 'X COST IS $###.##', #3, A
```

SECTION 6

MATRIX MANIPULATIONS

AND

MATRIX STATEMENTS

The BASIC statements discussed in the previous section permit the elements of a matrix to be defined and used as an element by element basis. The MAT statement, discussed in this section, allows matrices to be manipulated as a unit. In addition to the individual examples given in this section, examples showing the use of the MAT statement are given in Appendix A.

Although the arrays have a column number 0 and a row number 0, the MAT statement ignores all matrix elements that have one dimension equal to zero (i.e., the MAT statement manipulates vectors and matrices, 0 elements are indeterminate).

MATRIX REDIMENSIONING

The original bounds and the current bounds are determined by the DIM statement, or by the default bounds value (10) or (10,10), or by the first MAT statement that references a matrix. The current bounds of a matrix can be changed within certain constraints.

The total amount of storage defined by the current bounds must be less than or equal to the amount of storage set aside for the original bounds. For example:

```
100  DIM  A  (10, 10)
...

300  MAT  A = ZER (5, 5)

400  MAT  A = ZER (3, 24)

500  MAT  A = ZER (2, 29)
```

are all legal redimensions of the matrix A; but:

```
550  MAT  A = ZER (5, 25)
```

is not legal redimensioning of matrix A.

A matrix may be assigned the value of another matrix with different current bounds, provided this operation conforms to the rules for redimensioning just discussed. The current bounds of the target matrix are automatically changed to be the same as the current bounds of the matrix assigned.

When the current bounds of a matrix are changed, any elements of that matrix with one or more subscripts equal to 0 are destroyed.

INITIALIZATION STATEMENTS

There are three MAT statements to facilitate the assignment of the individual matrix elements.

Syntax

```
MAT  A  =  CON

or

MAT  A  =  IDN

or

MAT  A  =  ZER
```

where A is a numeric matrix.

These matrix initialization statements set the matrix specified to the left of the = to a constant matrix having the same bounds. The values to the right of the = are called matrix constants.

The constant CON sets each element of the matrix defined by matrix A to 1. Conversely, the constant ZER sets each element of the matrix defined by A to 0.

The constant IDN sets the matrix defined by matrix A to the identity matrix. This action is defined by the following algorithm:

```
A (I,J) = 1  IF  I = J

A (I,J) = 0  IF  I<> J
```

For the IDN assignment to be valid, the matrix A must be two-dimensional and the number of columns must equal the number of rows (i.e., A must be a square matrix).

Examples

```
200  MAT V = CON
```

sets elements of matrix V to all ones

```
300  MAT Z = ZER
```

sets elements of matrix Z to all zeroes

```
340  DIM I = (4,4)
400  MAT I = IDN
```

sets matrix I to the identity matrix
Elements of the matrix defined by matrix I are assigned as follows:

Row		Column	1	2	3	4
1	=		1	0	0	0
2	=		0	1	0	0
3	=		0	0	1	0
4	=		0	0	0	1

MATRIX INITIALIZATION WITH REDIMENSIONING

Matrices may also be redimensioned in the MAT...CON, MAT...ZER or MAT...IDN statments.

Syntax

```
MAT  A = CON (B1)

or

MAT  A = CON (B1, B2)

or

MAT  A = ZER (B1)

or

MAT  A = ZER (B1, B2)

or

MAT  A = IDN (B1, B1)
```

where A is a numeric matrix and B1 and B2 are expressions which define a matrix bound.

These matrix initialization statements set the matrix to the left of the = to a constant matrix having the bounds specified by B1 and B2; and in addition, assign values to the elements of the matrix defined by matrix A according to the functions of the specified MAT...ZER..., MAT...CON, and MAT...IDN statement.

Examples

```
20  DIM  X(4,5)

30  MAT  X = CON (3,3)

    X is   1   1   1

           1   1   1

           1   1   1

60  DIM  Y(3,3)

70  MAT  Y = Z ER (4,2)

    Y is   0   0

           0   0

           0   0

           0   0
```

MATRIX ASSIGNMENT

A matrix may be assigned the value of another matrix.

Syntax

```
MAT  A = B
```

where A and B are numeric matrices.

Both A and B must be either both one-dimensional (vectors) or both two-dimensional (matrices).

The matrix assignment statement sets the matrix appearing to the left of the = to the value of the matrix appearing to the right of the =. The current bounds of the target matrix are charged to the assigned matrix.

Examples

```
10  DIM A (6,6)

20  DIM B (5,4)

30  MAT A = B
```

the assignment at statement 30 is a legal assignment; but

```
15  DIM C - (10, 10)

25  DIM D - (2, 10)

35  MAT D = C
```

is not legal since the effect of the assignment is to try and assign a larger storage area, (matrix C) into the smaller one (matrix D) which would be charged with 80 more locations than were originally allocated.

MATRIX ADDITION

Syntax

```
MAT  A  =  B  +  C
```

where A, B, and C must all be either numeric vectors or numeric matrices. The elements of A are set to the sum of the corresponding elements of B and C. The matrices B and C must have the same current bounds; the bounds of the target matrix A are changed to the bounds of the input matrices (B and C).

Example

```
100  MAT X = Y + Z

220  MAT Y = X + Z
```

MATRIX SUBTRACTION

Syntax

```
MAT  A = B - C
```

where A, B, and C must all be either numeric vectors or numeric matrices. The matrix elements of A is set to the difference of the corresponding elements B and C. B and C must have the same current bounds, and the bounds of A are set to the current bounds of B and C.

Example

```
142  MAT X = Y - Z
```

MATRIX MULTIPLICATION

Matrix elements may be multiplied by scalar quantities or by elements of another matrix.

SCALAR MULTIPLICATION

Syntax

```
MAT  A = (E) * B
```

where A and B are numeric matrices and E is a numeric scalar expression.

This form of matrix multiplication sets the matrix A to the value of the product of each element of B times the value specified by E.

Matrices A and B must have the same number of dimensions. The current bounds of A are changed to the current bounds of B.

Examples

```
300  MAT  X  =  (5) * Y

320  MAT  X  =  (SQR(1-X/Y)) * B
```

PRODUCTS OF MATRICES

Syntax

```
MAT  X  =  Y  *  Z
```

X, Y, and Z are numeric two-dimensional matrices.

This form of matrix multiplication sets the matrix A to products of the matrices to the right of the =.

When two matrices are multiplied, the number of rows in the first matrix must equal the number of columns in the second matrix; the result is a matrix with the same number of columns as the first matrix and the same number of rows as the second matrix.

Examples

```
 10  DIM  A  (10, 10)
 20  DIM  B  (4, 5)
 30  DIM  C  (3, 3)
100  MAT  A  =  B  *  C
```

NOTE: While the statements of the form:

```
MAT  A = A + B
MAT  A = A - B
```

are allowed, the statement:

```
MAT  A = A * B
```

causes an error when the program is run.

TRANSPOSE OPERATIONS

Syntax

```
MAT  A  =  TRN  (B)
```

where A and B are either both numeric one-dimensional matrices or both numeric two-dimensional matrices.

To transpose statement sets the matrix A to the transpose of matrix B; the columns (rows) of A are the rows (columns) of B. The current bounds of A are changed. For example, if B is dimensioned M, N, the bounds of A are changed to N, M.

Example:

```
100  DIM  B  (5, 4)

110  MAT  A  =  TRN  (B)
```

A would be a matrix, the same as if it were fined by the statement:

```
MAT  A  =  INV  (B)
```

where A is a two-dimensional numeric matrix and B is a square two-dimensional numeric matrix.

The matrix A is set to the inverse of B. The bounds of A are set to the bounds of B.

Note that the statement:

```
A  =  INV(A)
```

is allowed by the Prime BASIC.

MAT READ

The MAT READ statement causes an entire matrix to be read (input).

Syntax

```
MAT  READ  A1  (D1, D2) ..., An (Dn, Dn)
```

where A1 ..., An are a list of numeric or string matrix names separated by commas, and D1 ... Dn are dimensions of the associated specified matrices. Specifying of dimensions D1 ... Dn are optional.

The MAT READ statement causes values from the data pool starting at the next available values, to be assigned, in order, to the matrix elements of the matrices specified.

Enough data values are read from the data pool to fill a matrix according to the current bounds of the matrix. If a matrix name in the MAT READ statement is followed by a bound list, the matrix is redimensioned to those bounds before any data is read.

Example

```
 10  DIM  A  (3,5)

 50  MAT  A  =  ZER

100  MAT  READ  A

200  DATA  1, 2, 3, 4, 5, 6, 7, 8, 9

210  DATA  10, 11, 12, 13, 14, 15
```

The statement at line 100 causes fifteen numbers to be read into matrix A by columns. For example: A(1,1) = 1; A (2,1) = 2, etc.

MAT READ FILE

The MAT READ statement causes a matrix to be read from an external data file and assigned, in order, to the matrix elements of the matrix specified.

Syntax

```
MAT  READ  #N, A1,..., An
```

where N is a file number (1-8) previously defined in a DEFINE FILE statement, and A1,...,An is a list of matrix names.

The file N consists of an ordered list of values that defines the contents of the elements of the matrix A. It may be created by a previous MAT WRITE FILE statement in the same or a previosuly executed program, or it may be created by the operating system editor.

Example:

```
10  DIM  V(10)

15  DEFINE FILE #1 = '(PTR)'

20  DIM  M(10, 20)

25  DEFINE FILE #2 = 'ARRAY'

30  MAT  READ #1, V

40  MAT  READ #2, M
```

The contents of the file #1 are read from the paper tape reader and assigned to the elements of the vector V. The contents of the file named ARRAY stored on the disk are read and assigned to the elements of matrix M.

MAT READ * FILE

Same as MAT READ file except the statement does not force a new record to be read. Any data remaining in a previous record are read as elements of the matrix.

MAT WRITE FILE

The MAT WRITE FILE statements causes a matrix to be written to an external data file.

Syntax

```
MAT WRITE  #N, AL, ..., An
```

where N is a file number (1-8) previously defined in a DEFINE FILE statement. If the output file is in ASCII (print) format, the character following matrix names in MAT WRITE FILE statements is used to control the spacing of the matrix elements in the output records. A comma specifies tabbed format and a colon specifies packed format. The optional character following the last matrix name controls the spacing of the elements of that last matrix and does not inhibit the termination of the last output read. A1, ..., An is a list of matrix names.

Example:

```
10  DEFINE FILE #1 = 'OUTPUT'
15  DIM  A  (100)
20  FOR  K = 1 TO 100
25  X = 2*3.1416
30  A(K) = X*K
40  NEXT K
50  MAT WRITE  #1, A
```

MAT INPUT

The MAT INPUT statement causes data values to be read from the terminal and assigned, in order, to the elements of a specified matrix.

Syntax

```
MAT  INPUT  A1, ..., An
```

where A1, ..., An is a list of matrix names separated by commas. The type of data provided must match the type of matrix being filled.

Example

```
10  DIM  B (5)
20  MAT  INPUT  B
```

allows information to be assigned to the elements of matrix B from the terminal. After the ? is printed, typing:

```
5, 10, 15, 20, 25
```

assigns those values to B(1) through B(5).

MAT PRINT STATEMENT

This statement causes an entire matrix to be printed.

Syntax

```
MAT  PRINT  A1, ..., An
```

where A1, ..., An is a list of matrix names separated by commas or colons.

The MAT PRINT statement causes all the elements of a matrix with subscripts that are not 0 to be printed column by column.

If a matrix name is followed by a colon, elements are printed with one space; otherwise, elements are printed in zoned format.

Example

```
100  DIM  M(2,6)
110  MAT  READ M
120  MAT PRINT M
200  DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
400  END
```

The above program yields the following output:

```
 1    2    3    4    5
 6    7    8    8   10
11   12
```

SECTION 7

INTERFACE CONVENTIONS

BASIC differs from compiler and assembly languages because its interpreter does not compile or assemble a reusable object text from the source program. Therefore, the BASIC interpreter must be present in high speed memory each time a user program is run. However, Prime BASIC provides the CALL statement to call FORTRAN or PMA (macro-assembly language) subroutines. Refer to Section 5 for details of the CALL statement format.

RELATING CALL TO SUBROUTINE

The user-supplied configuration file that is associated with the BASIC CALL statement is a table. The entries to this table are the addresses of PMA assembly language object text subroutines, or FORTRAN language object text subroutines, or a combination of both PMA and object text subroutines.

An example of two typical subroutines that may be called by a program written in BASIC is as follows:

```
*     SUBROUTINE TO START CLOCK
*
*
*
        ENT     STRTCK
        REL
*
STRTCK  DAC     **
        CRA
        STA     '61
        OCP     '20                 START CLOCK
        JMP*    STRTCK
        END


C       SUBROUTINE TO GET REAL TIME CLOCK WORD
C
C
        SUBROUTINE GETCLK(ARG)
        COMMON /LIST/ LIST(1)
C
        ARG=LIST(20'61)
        RETURN
        END
```

The configuration file is produced by modifying the CALLP source file on the MFD then assembling it. The following listing is the original source of the CALLP subroutine; immediately following that, is a listing that shows how the CALLP source was modified to call one PMA subroutine to start the real-time clock and one FORTRAN subroutine to get the real-time clock word.

```
*     CALLP (BASIC1/TRANSLTR) 24 JAN 74
*
*
*     ************************
*     *                      *
*     *   CALL PROCESSOR     *
*     *                      *
*     ************************
*
*-----CALL SUBROUTINE NUMBER IN BUF(1), ADDRESS OF ARGS IN BUF(2),
*     BUF(3), . . . , BUF(N).   (N THE NUMBER OR ARGS)
*
*
      SUBR  CALLP,CALL
      REL
*
*
$INSERT PCOMON
*
*
CALL  DAC   **
      LDX   BUF
      DRX
      NOP
      JMP   CAL1,1
*
*-----SUBROUTINE CALL LOCATIONS.
CAL1  JMP   S1
      JMP   S2
      JMP   S3
      JMP   S4
      JMP   S5
      JMP   S6
      JMP   S7
      JMP   S8
      JMP   S9
*
*-----DEFAULT SUBROUTINE CALLS.
```

```
S1    JMP    CAL2
S2    JMP    CAL2
S3    JMP    CAL2
S4    JMP    CAL2
S5    JMP    CAL2
S6    JMP    CAL2
S7    JMP    CAL2
S8    JMP    CAL2
S9    JMP    CAL2
*
*-----UNIMPLEMENTED SUBROUTINE CALL.
CAL2  CALL   RERROR
      DAC    =C'BC'
*
      END
```

The "Modified" CALLP follows:

```
*     CALLP (BASIC1/TRANSLTR) 24 JAN 74
*
*
*     ****************************
*     *                          *
*     *    CALL PROCESSOR        *
*     *                          *
*     ****************************
*
*-----CALL SUBROUTINE NUMBER IN BUF(1), ADDRESS OF ARGS IN BUF(2),
*     BUF(3), . . . , BUF(N).   (N THE NUMBER OR ARGS)
*
*
      SUBR  CALLP,CALL
      REL
*
*
$INSERT PCOMON
*
*
CALL  DAC   **
      LDX   BUF
      DRX
      NOP
      JMP   CAL1,1
*
*-----SUBROUTINE CALL LOCATIONS.
CAL1  JMP   S1
      JMP   S2
      JMP   S3
      JMP   S4
      JMP   S5
      JMP   S6
      JMP   S7
      JMP   S8
      JMP   S9
*
*-----DEFAULT SUBROUTINE CALLS.
```

```
S1    CALL  STRTCK              START CLOCK
      JMP*  CALL                RETURN
*
S2    LDA   BUF+1               INSURE 1 ARGUMENT
      S1A
      SZE
      JMP   CAL2                NOT 1 ARG, ERROR
      CALL  GETCLK
      DAC*  BUF+2               1 ARG
      JMP*  CALL
*
S3    JMP   CAL2
S4    JMP   CAL2
S5    JMP   CAL2
S6    JMP   CAL2
S7    JMP   CAL2
S8    JMP   CAL2
S9    JMP   CAL2
*
*----UNIMPLEMENTED SUBROUTINE CALL.
CAL2  CALL  RERROR
      DAC   =C'BC'
*
      END
```

MODIFYING COMMAND FILE

The source of the modified CALLP listed above is assembled. Then, the command file must be modified. Depending on which version of BASIC the user desires to use, one of the following command files must be modified:

C ←BASC	*BASIC with no PRINT USING OR MAT statements
C←BUSE	*BASIC with PRINT USING statement
C←BMAT	*BASIC with MAT statements
C←BALL	*BASIC with both PRINT USING and MAT statements

Any or all of the above command files are modified using the editor, ED, as follows:

1. Locate the command line:

   LOAD B←CALL

2. Insert LOAD commands for the subroutines to be called by the program(s) written in BASIC. For example, to call the subroutines listed in the sample modification of the CALLP subroutine, the following statements are inserted:

   LOAD B←STRT

   LOAD B←GTCLK

NOTE: The above 'LOAD' is a command to the DOS/DOS-VM loader; not to be confused with the BASIC command of the same name.

RUNNING PROGRAM WITH CALL STATEMENTS

After inserting the proper LOAD commands, execute the command file and save the results (see the Program Development Software User Guide). At this time, the desired version of BASIC, the modified CALLP subroutine, and the called subroutines are loaded as an entity so that programs written in BASIC may call the designated subroutines. For example, the following program starts the real-time clock and prints a clock value every 300 microseconds.

```
>LIST
  10  CALL 1
  20  I=300
  30  CALL_2(J)
  40  IF I<>J GOTO 30
  50  PRINT I,
  60  I=I+300
  70  GOTO 30
```

Sample Output:

```
>RUN
300             600             900             1200            1500
1800            2100            2400            2700            3000
>
```

APPENDIX A

SAMPLE PROGRAMS

```
 10 REM ******    EXAMPLE 1    ******
 20 REM
 30 REM PROGRAM TO CALCULATE MILES PER GALLON AND PRINT REPORT.
 40 REM
 50 REM THIS PROGRAM GIVES AN EXAMPLE OF HOW A BASIC PROG
 60 REM DATA AND DO SIMPLE CALCULATIONS AND USE THE
 70 REM STATEMENT TO PRINT A REPORT.
 80 REM
 90 REM IT IS INTENDED TO GIVE THE USER AN IDEA OF WHAT IS TO
100 REM SOLVE A PROBLEM AND
110 REM BASIC STATE
120 REM
130 REM     OF THE STATEMENTS USED ARE:
140 REM     LET (ASSIGNMENT), PRINT, AND DATA
150 PRINT 'DATE','ODOMETER','MILES','GALLONS','    MPG'
160 REM NOW INITIALIZING SOME VARIABLES USED LATER.
170 LET M1=0
180 LET N1=0
190 LET G1=0
200 LET K1=0
210 READ K
220 REM K IS ODOMETER READING.
230 LET K1=K
240 REM K1 IS SET TO ORIGINAL ODOMETER READING.
250 READ N
260 REM N IS SET TO LATER ODOMETER READINGS.
270 REM N IS ALSO USED AS A FLAG TO TERMINATE LOOP.
280 REM STATEMENT 140 IS ENTRY INTO LOOP.
290 IF N=0 THEN 480
300 REM WHEN N<> 0 LOOP CONTINUES.
310 READ D
320 REM D IS DATE.
330 READ G
340 REM G IS GALLONS USED SINCE LAST ODOMETER READING.
350 LET M=N-K
360 REM M IS INCREMENT OF TOTAL MILAGE.
370 M1=M1+M
380 REM M1 IS RUNNING TOTAL OF MILES.
```

```
390 LET A=M/G
400 REM CALCULATION OF MILES PER GALLON.
410 PRINT D,N,M,G,A
420 LET K=N
430 REM UPDATES K TO LAST ODOMETER READING.
440 LET G1=G1+G
450 REM G1 IS TOTAL GALLONS USED
460 GOTO 250
470 REM CONTINUE LOOP UNTIL N = 0.
480 PRINT 'TOTAL',' ',M1,G1,(K-K1)/G1
490 REM NOTE USE OF ',' TO SKIP ONE PRINT FIELD.
500 DATA 45882,46193
510 DATA 21574,16.8,46315
520 DATA 22274,9.4,46505
530 DATA 30174,12.7,46855
540 DATA 30874,17.6,47067
550 DATA 31574,15.2,47314
560 DATA 32274,14.7,47464
570 DATA 32974,10.6,0
580 DATA 0
590 STOP
```

The following is the output from Example 1:

```
LOAD 'EXAMP1'
DATE          ODOMETER      MILES         GALLONS          MPG
21574         46193         311           16.8          18.5119
22274         46315         122           9.4           12.9787
30174         46505         190           12.7          14.9606
30874         46855         350           17.6          19.8864
31574         47067         212           15.2          13.9474
32274         47314         247           14.7          16.8027
32974         47464         150           10.6          14.1509
TOTAL                       1582          97            16.3093

STOPPED AT LINE 350
```

```
 10 REM *****   EXAMPLE 2   *****
 20 REM
 30 REM THE FOLLOWING STATEMENTS ARE A MODIFICATION OF EXAMPLE 1
 40 REM TO ALLOW DATA TO BE READ FROM A FILE RATHER THAN DATA
 50 REM STATEMENTS.
 60 REM
 70 DEFINEFILE# 1='GASDAT'
 80 PRINT 'DATE', 'ODOMETER', 'MILES', 'GALLONS', 'MILES/GALLON'
 90 LET M1=0
100 LET N1=0
110 LET G1=0
120 LET K1=0
130 LET K=45882
140 REM MUST SET INITIAL VALUE OF K BY HAND,
150 REM OR READ IT IN FROM ANOTHER FILE TO PREVENT RECORD FROM BEING
160 REM DISCARDED.
170 LET K1=K
180 READ# 1,N,D,G
190 REM ALL ITEMS MUST BE READ OR REST OF RECORD IS DISCARDED.
200 IF N=0 THEN 280
210 LET M=N-K
220 M1=M1+M
230 LET A=M/G
240 PRINT D,N,M,G,A
250 LET K=N
260 LET G1=G1+G
270 GOTO 180
280 PRINT 'TOTAL',' ',M1,G1,(K-K1)/G1
290 STOP
```

The following is a sample of the output from Example 2. The results are similar to Example 1 and are included for comparison.

```
LOAD "EXAMP2"

RUN
DATE          ODOMETER      MILES         GALLONS       MILES/GALLON
21574         46193         311           16.8          18.5119
22274         46315         122           9.4           12.9787
30174         46505         190           12.7          14.9606
30374         46855         350           17.6          19.8864
31574         47067         212           15.2          13.9474
32274         47314         247           14.7          16.8027
32974         47464         150           10.6          14.1509
TOTAL                       1582          97            16.3093

STOPPED AT LINE 350
```

```
  5 REM *****   EXAMPLE 3 *****
 10 REM THIS PROGRAM CALCULATES THE FACTORS OF A POSITIVE
 15 REM INTEGER BETWEEN 1 AND 999999 INCLUSIVE.
 20 REM
 25 REM IF THE NUMBER HAS NO FACTORS, A MESSAGE IS RETURNED
 30 REM THAT THE NUMBER IS PRIME.  CHECKING IS MADE FOR SOME
 35 REM SPECIAL CASES.
 40 REM
 45 REM THE PROGRAM SHOWS THE US OF GOSUB AND RETURN STATEMENTS
 50 REM TO PRODUCE BOTH NESTED AND SEQUENTIAL SUBROUTINES
 55 REM IN A PROGRAM.  IT ALSO DEMONSTRATES THE USE OF THE PRINT
 60 REM AND THE INPUT STATEMENTS TO PRODUCE AN INTERACTIVE CON-
 65 REM VERSATIONAL PROGRAM.
 70 REM
 75 PRINT 'PLEASE TYPE YOUR NUMBER:';
 80 INPUT A
 85 LET P=1
 90 LET S=1
 95 LET H=1
100 REM H INITIALIZING FLAGS.
105 GOSUB 375
110 GOSUB 350
115 PRINT 'IS THIS YOUR LAST NUMBER:';
120 INPUT A$
125 IF A$<>'YES' THEN 75
130 END
135 REM ********END OF MAIN PROGRAM.
140 REM SUBROUTINE TO CHECK IF A IS NOT AN INTEGER.
145 IF A=INT(A) THEN 160
150 PRINT 'NUMBER MUST BE AN INTEGER.'
155 GOSUB 500
160 RETURN
165 REM SUBROUTINE RETURNS MESSAGE IF A > 999999
170 IF A<=999999 THEN 185
175 PRINT 'SORRY, AT PRESENT, NUMBER MAY NOT EXCEED 999999.'
180 GOSUB 500
185 RETURN
190 REM SUBROUTINE TO HANDLE A = 1
```

```
195 IF A<>1 THEN 215
200 PRINT A;'IS 1 AND IS DIVISIBLE BY ONLY ITSELF AND 1.'
205 PRINT 'HOWEVER, IT IS NOT A PRIME NUMBER.'
210 GOSUB 500
215 RETURN
220 REM SUBROUTINE TO CHECK IF A IS 0 OR NEGATIVE.
225 IF A>=1 THEN 245
230 PRINT 'NUMBER MAY NOT BE EITHER ZERO OR A NEGATIVE VALUE.'
235 PRINT 'YOUR NUMBER';A;'IS INVALID.'
240 GOSUB 500
245 RETURN
250 REM SUBROUTINE TO HANDLE A = 2,3.
255 IF A>3 THEN  IF A<2 GOTO 270
260 GOSUB 350
265 GOSUB 500
270 RETURN
275 REM SUBROUTINE TO PRINT HEADER LINE.
280 IF P=1 THEN 310
285 IF H=0 THEN 310
290 PRINT 'NUMBER';A;'IS DIVISIBLE BY:'
295 PRINT
300 PRINT A,'AND','1',' ',' ',' '
305 LET H=0
310 RETURN
315 REM SUBROUTINE THAT PRINTS DIVISORS AND QUOTIENTS.
320 IF P=1 THEN 345
325 PRINT X,'AND',Q2,' ',' ',' '
330 IF X<>2 THEN 345
335 PRINT '1','AND',A
340 PRINT
345 RETURN
350 REM SUBROUTINE TO PRINT MESSAGE WHEN NUMBER IS A PRIME.
355 IF P=0 THEN 370
360 PRINT 'NUMBER';A;'IS A PRIME NUMBER.'
365 PRINT
370 RETURN
375 REM PERFORM CALCULATION AND GET ROUTINES TO PRINT RESULTS.
380 IF S=0 THEN 420
```

```
385 FOR X=INT(A/2) TO 2 STEP -1
390 LET Q1=A/X
395 LET Q2=INT(A/X)
400 LET R=Q1-Q2
405 GOSUB 425
410 NEXT X
415 GOSUB 660
420 RETURN
425 REM CALL PRINT ROUTINES ETC.
430 IF R<>0 THEN 450
435 LET P=0
440 GOSUB 275
445 GOSUB 315
450 RETURN
455 REM SUBROUTINE TO ENTER SPECIAL ROUTINES.
460 REM IF A IS NEGATIVE OR ZERO, OR 1,2,3, OR IF A IS A FRACTION
465 GOSUB 140
470 GOSUB 165
475 IF A>3 THEN 495
480 GOSUB 190
485 GOSUB 220
490 GOSUB 250
495 RETURN
500 PRINT
505 LET S=0
510 RETURN
```

The following is same sample output from Example 3.

```
LOAD 'EXAMP3'
RUN

PLEASE TYPE YOUR NUMBER: 0
NUMBER MAY NOT BE EITHER ZERO OR A NEGATIVE VALUE.

YOUR NUMBER 0 IS INVALID.
IS THIS YOUR LAST NUMBER: NO
PLEASE TYPE YOUR NUMBER: 1
1 IS 1 AND IS DIVISIBLE BY ONLY ITSELF AND 1.
HOWEVER, IT IS NOT A PRIME NUMBER.

IS THIS YOUR LAST NUMBER: NO
PLEASE TYPE YOUR NUMBER: 2
NUMBER 2 IS A PRIME NUMBER.
```

Sample output from Example 3 (cont)

```
IS THIS YOUR LAST NUMBER: NO
PLEASE TYPE YOUR NUMBER: 4
NUMBER 4 IS DIVISIBLE BY:

4               AND             1
2               AND             2
1               AND             4

IS THIS YOUR LAST NUMBER: NO
PLEASE TYPE YOUR NUMBER: 17
NUMBER 17 IS A PRIME NUMBER.


IS THIS YOUR LAST NUMBER: NO
PLEASE TYPE YOUR NUMBER: 3245673901234567 89
SORRY, AT PRESENT, NUMBER MAY NOT EXCEED 999999.

..
PLEASE TYPE YOUR NUMBER: 56
NUMBER 56 IS DIVISIBLE BY:

56              AND             1
28              AND             2
14              AND             4
8               AND             7
7               AND             8
4               AND             14
2               AND             28
1               AND             56

IS THIS YOUR LAST NUMBER: YES

END AT LINE 230
```

```
100  REM *****   EXAMPLE 4   *****
110  REM
120  REM THIS PROGRAM USES THE ARRAY PROCESSING CAPABILITIES
130  REM OF BASIC TO COMPUTE THE TOTAL DOLLAR VALUE OF THREE
140  REM PRODUCTS SOLD BY FIVE SALESMEN.
150  REM
160  REM VECTOR ELEMENT P(M) IS THE PRICE OF THE N-TH PRODUCT.
170  REM MATRIX ELEMENT S(M,N) IS THE TOTAL NUMBER OF THE M-TH
180  REM PRODUCT SOLD BY THE N-TH SALESMAN.
190  REM
200  REM
210  DIM S(6,4)
220  DIM P(4)
225  T=0
230  MAT P= CON
240  MAT S= ZER
250  READ P(1),P(2),P(3)
260  READ S(1,1),S(1,2),S(1,3),S(2,1),S(2,2),S(2,3)
270  READ S(3,1),S(3,2),S(3,3),S(4,1),S(4,2),S(4,3)
280  READ S(5,1),S(5,2),S(5,3)
290  FOR I=1 TO 5
300  FOR J=1 TO 3
310  S(I,J)=P(J)*S(I,J)
311  T=S(I,J)+T
320  PRINT I;J,S(I,J)
330  NEXT J
340  NEXT I
341  PRINT
342  PRINT
350  MATPRINT S
351  PRINT
352  PRINT "TOTAL SALES =";T
360  DATA 1.29,2.54,5.48
370  DATA 47,24,16,56,38,12,76,23,14,76,45,12,45,34,23
380  STOP
```

The following is sample output from Example 4.

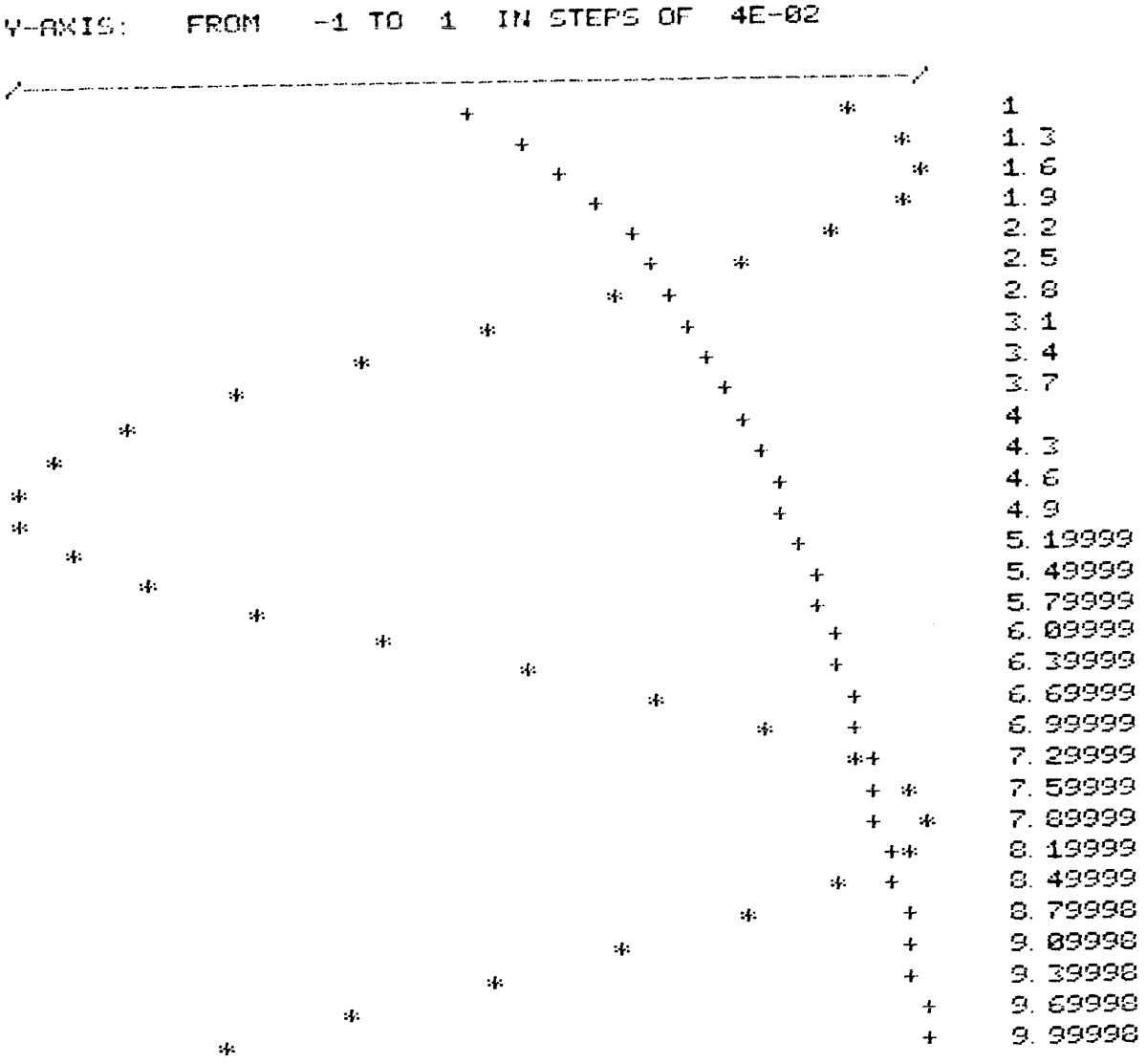
```
1 1             60.63
1 2             60.96
1 3             87.68
2 1             72.24
2 2             96.52
2 3             65.76
3 1             98.04
3 2             58.42
3 3             76.72
4 1             98.04
4 2             114.3
4 3             65.76
5 1             58.05
5 2             36.36
5 3             126.04
60.63           72.24           98.04           98.04           58.05
0               60.96           96.52           58.42           114.3
36.36           0               87.68           65.76           76.72
65.76           126.04          0               0               0
0               0               0               0

TOTAL SALES = 1225.52

STOPPED AT LINE 380
```

```
100 REM *****   EXAMPLE 5   *****
110 REM
120 REM MULTIPLE PLOT PROGRAM
130 REM
140 REM
150 DEF FNF(X)=SIN(X)
160 DEF FNG(X)=LOG(X)/LOG(10)
170 READ A,B,S
180 READ C,D,N
190 LET H=(D-C)/N
200 IF N<=50 THEN 230
210 PRINT " ONLY 50 SUBDIVISIONS ALLOWED ON Y-AXIS"
220 STOP
230 DEF FNR(X)=INT(X+.5)
240 PRINT "Y-AXIS:   FROM  ";C;"TO ";D;" IN STEPS OF ";H
250 PRINT
260 L$="/"
270 FOR I=1 TO N-1
280 L$=L$+"-"
290 NEXT I
300 PRINT L$+"/"
310 FOR X=A TO B STEP S
320 LET Y=FNF(X)
330 LET Y1=FNR((Y-C)/H)
340 LET Y=FNG(X)
350 LET Y2=FNR((Y-C)/H)
360 L$=""
370 FOR I=0 TO N
380 IF I=Y1 THEN 420
390 IF I=Y2 THEN 440
400 L$=L$+" "
410 GOTO 450
420 L$=L$+"*"
430 GOTO 450
440 L$=L$+" "
450 NEXT I
460 PRINT L$+"   ";X
470 NEXT X
480 DATA 1,10,.3
490 DATA -1,1,50
500 DATA 0
510 STOP
```

The following is sample output from Example 5.



```
100 REM ******   EXAMPLE 6   ******
110 REM FIBONACCI NUMBER GENERATOR
120 REM
130 INPUT E
140 I=1
150 J=2
160 PRINT I,J,
170 FOR K=1 TO E
180 F=I+J
190 I=J
200 J=F
210 PRINT F,
220 NEXT K
230 END
```

Sample output:

```
!20
1             2             3             5             8
13            21            34            55            89
144           233           377           610           987
 1597          2584          4181          6765          10946
 17711         28657
END AT LINE 230
```

```
100 REM *****   EXAMPLE 7   *****
110 REM
120 REM LUNAR LANDING PROGRAM
130 REM
140 PRINT
150 PRINT 'CONTROL CALL LUNAR MODULE.  MANUAL CONTROL IS NECESSARY.'
160 PRINT 'YOU MAY RESET FUEL RATE K EACH 10 SECS TO 0 OR ANY VALUE'
170 PRINT 'BETWEEN 8 & 200 LBS/SEC.  YOUVE 16000 LBS. FUEL. ESTIMMATED'
180 PRINT 'FREE-FALL IMPACT TIME = 120 SECS.  CAPSULE WEIGHT = 32500 LBS'
190 B$='####    ####   #####        ----#.###       #####.###'
200 PRINT 'FIRST RADAR CHECK COMING UP.'
210 PRINT
220 PRINT
230 PRINT
240 PRINT 'COMMENCE LANDING PROCEDURE'
250 PRINT
260 PRINT 'TIME      ALTITUDE','VELOCITY','FUEL (LBS)','FUEL RATE'
270 PRINT 'SECS     MILES  FEET','   MPH'
280 PRINT
290 L=0
300 A=120
310 V=1
320 M=32500
330 N=16500
340 G=1E-03
350 Z=1.8
360 PRINTUSING B$,L,INT(A):INT(5280*(A-INT(A))),3600*V,M-N,
370 PRINT 'K=':
380 INPUT K
390 T=10
400 ON SGN(K)+2 GOTO 430,460,410
410 IF K<8 GOTO 430
420 IF K<=200 GOTO 460
430 PRINT 'NOT POSSIBLE'
440 PRINT TAB(56):
450 GOTO 370
460 IF M-N<1E-03 GOTO 570
470 IF T<1E-03 GOTO 360
480 S=T
490 IF (N+S*K)<=M GOTO 510
500 S=(M-N)/K
510 GOSUB 1020
520 IF I<=0 GOTO 900
530 IF V<=0 GOTO 550
540 IF J<0 GOTO 950
550 GOSUB 840
560 GOTO 460
570 PRINT 'FUEL OUT AT':L:'SECS'
580 S=(-V+SQR(V*V+2*A*G))/G
590 V=V+G*S
```

```
 600 L=L+S
 610 PRINT "ON THE MOON AT":L:"SECS"
 620 W=3600*V
 630 PRINT "IMPACT VELOCITY OF":W:"MPH"
 640 PRINT "FUEL LEFT IS":M-N:"LBS"
 650 IF W>1 GOTO 680
 660 PRINT "PERFECT LANDING!"
 670 GOTO 780
 680 IF W>10 GOTO 710
 690 PRINT "GOOD LANDING"
 700 GOTO 780
 710 IF W>25 GOTO 740
 720 PRINT "POOR LANDING"
 730 GOTO 780
 740 IF W>60 GOTO 770
 750 PRINT "CRAFT DAMAGED; GOOD LUCK TO YOU AND THE RED SOX."
 760 GOTO 780
 770 PRINT "FATAL CRASH; NO SURVIVORS"
 780 PRINT
 790 PRINT "TRY AGAIN?":
 800 INPUT A$
 810 IF A$="YES" GOTO 200
 820 IF A$="NO" THEN  STOP
 830 GOTO 790
 840 L=L+S
 850 T=T-S
 860 M=M-S*K
 870 A=I
 880 V=J
 890 RETURN
 900 IF S<5E-03 GOTO 610
 910 S=2*A/(V+SQR(V*V+2*A*(G-Z*K/M)))
 920 GOSUB 1020
 930 GOSUB 840
 940 GOTO 900
 950 W=(1-M*G/(Z*K))/2
 960 S=M*V/(Z*K*(W+SQR(W*W+V/Z)))+5E-02
 970 GOSUB 1020
 980 ON SGN(I)+2 GOTO 900,900,990
 990 GOSUB 840
1000 ON SGN(J)+2 GOTO 1010,460,460
1010 ON SGN(V)+2 GOTO 460,460,950
1020 Q=S*K/M
1030 Q1=0
1040 FOR Q2=9 TO 1 STEP -1
1050 Q1=Q*(1/Q2+Q1)
1060 NEXT Q2
1070 J=V+G*S-Z*Q1
1080 Q1=0
1090 FOR Q2=9 TO 1 STEP -1
1100 Q1=Q*(Q1+1/(Q2*(Q2+1)))
1110 NEXT Q2
1120 I=A-G*S*S/2-V*S+Z*S*Q1
1130 RETURN
```

Sample output from Example 7.

```
COMMENCE LANDING PROCEDURE

TIME      ALTITUDE            VELOCITY      FUEL (LBS)     FUEL RATE
SECS      MILES   FEET          MPH

   0      120        0       3600.000      16000.000      K= !0
  10      109     5015       3636.000      16000.000      K= !0
  20       99     4223       3672.000      16000.000      K= !0
  30       89     2903       3708.000      16000.000      K= !0
  40       79     1055       3744.000      16000.000      K= !0
  50       68     3959       3780.000      16000.000      K= !0
  60       58     1055       3816.000      16000.000      K= !0
  70       47     2903       3852.000      16000.000      K= !200
  80       37     1929       3476.430      14000.000      K= !200
  90       28     1384       3072.940      12000.000      K= !200
 100       20     1706       2637.460      10000.000      K= !200
 110       13     3399       2164.970       8000.000      K= !200
 120        8     1772       1649.140       6000.000      K= !200
 130        4     2795       1081.920       4000.000      K= !200
 140        2     2018        452.719       2000.000      K= !40
 150        1     1429        347.074       1600.000      K= !60
 160        0     2943        164.626       1000.000      K= !40
 170        0     1364         50.793        600.000      K= !8
 180        0      577         56.406        520.000      K= !20
 190        0       47         15.310        320.000      K= !19.4
ON THE MOON AT 193.354 SECS
IMPACT VELOCITY OF .813765 MPH
FUEL LEFT IS 245.219 LBS
PERFECT LANDING!
```

```
 10  REM *****   EXAMPLE 8   *****
 20  REM
 30  REM THIS PROGRAM SEARCHES A DATA FILE CONSISTING OF
 40  REM STRING ITEMS AND RETURNS THE LINE (RECORD SEQUENCE)
 50  REM NUMBER THAT DEFINES THE LOCATION OF THE STRING(S).
 60  REM
 70  PRINT 'ENTER FILENAME',
 80  INPUT F$
 90  DEFINEFILE# 1=F$
100  PRINT 'ENTER STRING',
110  INPUT S$
111  C=1
120  IF S$='NO MORE' THEN 260
130  READ# 1, B$
140  FOR I=1 TO LEN(B$)
150  IF B$='EOT' THEN 200
160  IF B$=S$ THEN 230
170  IF S$=SUB(B$, I, LEN(S$)) THEN 230
171  C=C+1
180  NEXT I
190  IF B$<>S$ THEN 130
200  PRINT S$:'NOT FOUND.
210  REWIND# 1
220  GOTO 100
230  PRINT S$:'FOUND AT CHARACTER POSITION':C:'.'
240  REWIND# 1
250  GOTO 100
260  END
```

Output from Example 8.

```
>RUN
ENTER FILENAME                !SOURCE
ENTER STRING  !GORGE
GORGE FOUND AT CHARACTER POSITION 63
ENTER STRING  !AARDVAARK
AARDVAARK FOUND AT CHARACTER POSITION 1
ENTER STRING  !SYZYGY
SYZYGY FOUND AT CHARACTER POSITION 73
ENTER STRING  !XXXX
XXXX NOT FOUND.
ENTER STRING  !ADZE
ADZE FOUND AT CHARACTER POSITION 10
ENTER STRING  !NO MORE

END AT LINE 260

>
```

```
 10  REM *****   EXAMPLE 9   *****
 12  REM
 14  REM THIS PROGRAM SIMULATES AN N DIMENSIONAL ARRAY.
 16  REM
 20  DIM V(1000)
 30  INPUT D1,D2,D3
110  FOR I=1 TO D1
111  FOR J=1 TO D2
112  FOR K=1 TO D3
115  X=(I-1)*D2+(J-1)*D3+K
120  V(X)=I+2*J+K^2
130  PRINT V(X),
140  NEXT K
150  PRINT
160  NEXT J
170  PRINT
180  NEXT I
200  STOP
```

Output from Example 9.

```
>RUN
?2,3,4
4             7             12            19
6             9             14            21
8             11            16            23

5             8             13            20
7             10            15            22
9             12            17            24
```

APPENDIX B

ERROR MESSAGES DEFINITIONS

CODE	S/E	REASON
AD	S	OPERAND ERROR
AO	E	STORAGE SPACE EXCEEDED (OCCURS DURING ARRAY ALLOCATION)
AR	S	A(OR A$(EXPECTED (E.G. DIM A$(...,); MAT STMT ARRAY NAME ERROR
BD	E	MATRIX DIMENSIONING ERROR
BD	S	MAT DIM HAS IMPROPER FORMAT
BE	F	PRINT/WRITE USING EXPR/FORMAT TYPE MISMATCH (STRING EXPR IN LIST WITH ARITH FORMAT; OR ARITH EXPR IN LIST WITH STRING FORMAT)
BL	S	STMT # + LOAD BIAS > 9999; STMT # NOT FOUND WHERE EXPECTED
BP	E	RECORD # IN POSITION <1 OR UNIT NOT BIN DA DISK
BU	E	FILE I/O UNIT NUMBER NOT 1-8
CH	S	ILLEGAL CHARACTER
CN	S	CONSTANT EXPECTED
CP	S	READ/WRITE UNIT IMPROPER
CR	S	CHARACTER NOT EXPECTED
CV	S	V= EXPECTED (E.G. FOR I=...)
DF	E	DEFINE FILE ERROR - - RECORD SIZE SPECIFIED <2 OR >512 - NO STORAGE FOR RECORD BUFFER - ERROR OPENING UNIT; DRIVER MESSAGE IS PRINTED (NOTE: A DF ERROR CAN ALSO OCCUR ON A LOAD/FILE STATEMENT IF OPEN ERROR OCCURS)
DM	S	DEFINE FILE MODE SPECIFIER ERROR
EE	S	EXPRESSION NOT OF TYPE EXPECTED
ES	S	STRING EXPRESSION EXPECTED
EX	S	ON END UNIT # EXPR IMPROPER
FE	E	PRINT/WRITE USING FORMAT ERROR
FM	E	FOR-NEXT MATCHING ERROR
FN	E	UNDEFINED FN FUNCTION
FN	S	FN FUNCTION NAME NOT A-Z
FO	E	STORAGE SPACE EXCEEDED (OCCURS DURING STRING OPERATIONS OR FOR STMT)
FP	S	FUNCTION NAME NOT FOLLOWED BY (

where E/S means Source (S) or Execution (E).

FT	S	FOR SEPARATOR ERROR
GO	E	MORE THAN 10 OUTSTANDING GOSUB'S
GT	S	GOTO EXPECTED
IC	E	ARRAY DIMENSION < 1
ID	S	UNRECOGNIZED STATEMENT
IE	E	ILLEGAL MAT MPY EXPR (E.G. MAT A=B*A)
IO	S	INTEGER > 32767
IS	S	IO EXPECTED IN POSITION STMT
IT	S	EXPRESSION FORMAT
LG	E	LOG ARGUMENT <= 0
LT	S	READ/INPUT LIST ERROR
ML	S	DEF NOT FOLLOWED BY FNX (
MM	S	MIXED MODE LET (E.G. I=A$; A$=I)
MR	S	V) NOT FOUND IN DEF STMT
MS	S	MIXED ARITH + STG ITEMS IN EXPR
OV	E	ON EXPRESSION OUT OF RANGE
PN	S	RIGHT PARENTHESIS REQUIRED
PO	S	EXPRESSION PARENTHESIS > 10 DEEP
PR	S	ONLY ONE ARGUMENT IN SUB FUNCTION REFERENCE
RE	S	MIXED STRING & ARITHMETIC ITEMS IN EXPRESSION
RI	E	# BASIC USES AS INTEGER > 32767; OCCURS IN - - SUBSCRIPT/DIMENSION - I/O STATEMENT UNIT # - RECORD SIZE IN DEFINE FILE STMT - ON STATEMENT EXPRESSION - TAB FUNCTION ARGUMENT - SUB FUNCTION NUMERIC ARGUMENTS - SUBROUTINE # IN CALL STATEMENT
SC	S	STRING CONSTANT NOT ALLOWED
SE	E	ARITHMETIC OVERFLOW
SF	F	EXPRESSION STACK FULL
SI	S	STRING ITEM NOT ALLOWED OR ILLEGAL OP IN STRING EXPR
SM	E	MATRIX NOT INVERTABLE
SN	E	UNDEFINED STATEMENT NUMBER
SQ	E	SQR FUNCTION ARGUMENT < 0
SS	E	SUBSCRIPT OUT OF RANGE OR WRONG # OF SUBSCRIPTS
ST	S	UNTERMINATED STRING CONSTANT
~~S~~	E	DIVIDE BY ZERO
Z	E	DIVIDE BY ZERO
0	S	STORAGE SPACE FOR PROGRAM EXCEEDED
R	E	RETURN EXECUTED WITHOUT OUTSTANDING GOSUB
F	E	UNIT REFERENCED BY I/O STMT NOT DEFINED
E	E	WRITE # ERROR; DRIVER MESSAGE IS PRINTED (ERROR IS NOT RECOVERABLE; BASIC MARKS THE UNIT INDEFINED)
P	E	WRITE # TO UNIT DEFINED BY DEFINE READ FILE STMT
R	E	READ AFTER WRITE TO NON DA UNIT
0	E	READ/INPUT # EXCEED $10^{-}(+/-38)$
0	S	CONSTANT EXCEEDS $10^{-}(+/-38)$
X	E	SIN;COS;EXP ARGUMENT > 32767

APPENDIX C

BASIC SUMMARY

Specification Statements:

```
REM   Comment Line
DIM   A (3), B (40,3)
DEF   FNX(I) = 2/COS(I) *3
TRACE ON
TRACE OFF
BREAK ON 40,318,215,10
BREAK OFF 10,40
BREAK OFF
```

Definition Statements:

```
LET I3=SIN(K-4.5)+Q3
I3 = SIN (X-4.5)+Q3
B$ = '0001'
LET  S$(J+5) = M$+D$+'.00'
```

Input/Output Statements:

```
DATA  2,3,4, - 3.7E2
RESTORE
READ  A1, A2, A3
READ  #3, A1, A2, A3
DEFINE FILE #3 ='TEST 3'
DEFINE FILE #(I+3) ='(LPR)'
REWIND #3
INPUT I3, I1, X(1,3)
PRINT  X4, 'FEET'
WRITE  #3, X4, 'FEET'
PRINT USING F$, X1, X4
WRITE USING F$, #3, X1, X4
ON END #1 GO TO 999
```

Formatted Print Descriptors:

#	Replace with Digit
.	Insert Dec. Point
,	Insert comma if needed
↑↑↑↑	Insert exponent field
+	Insert (+) or (-)
-	Insert (SP) or (-)
++	Insert leading (+) or (-)
--	Insert leading (SP) or (-)
$	Insert dollar sign.
$$	Insert leading dollar sign.

Control Statements:

```
GO TO 50
GO SUB 30
RETURN
IF  C1< 1  GO TO 40
If  D4 = 'ANY' THEN 50
IF  X  = 5 THEN Z = 3
FOR  C1 = 2 TO STEP 1
FOR  C1 = 2, 10, 1
FOR  A4 = 50, -4.5, -1.2
NEXT  A4
ON (I-1) GO TO 10, 20, 60
STOP
END
CALL  5 (A3, 6, I-2)
CALL  1
```

Matrix Statements:

```
MAT  X  =  ZER
MAT  X  =  CON
MAT  X  =  IDN
MAT  X  =  Y + Z
MAT  X  =  Y - Z
MAT  X  =  4 * Z
MAT  X  =  (5) * Y
MAT  X  =  TRN (Y)
MAT  X  =  INV (Y)
MAT  READ  X, Y, Z
MAT  READ  #N,X, Y, Z
MAT  WRITE  #N,X, Y, Z
MAT  INPUT  X, Y, Z
MAT  PRINT  X, Y, Z
```

Functions:

```
SIN(X)
COS(X)
TAN(X)
ATN(X)
LOG(X)
EXP(X)
SQR(X)
ABS(X)
SGN(X)
INT(X)
RND(X)
LEN(X$)
SUB(X$,Y,Z)
```

Arithmetic Operators

+	ADD
-	SUB
*	MUL
/	DIV
↑	EXPON

Relational Operators

<	.LT
>	.GT
=	.EQ
<= =<	.LE
>= =>	.GE
<> ><	.NE

String Operator

+	Concatenation

System Commands:

```
LOAD'ALPHA'          LOAD'BETA', 1000
FILE'PNAME'          FILE'PNAME', 1000, 1999
LIST                 LIST 1000, 1999
RUN                  RUN 45
NEW
CLEAR
CONTINUE
QUIT
```

APPENDIX D

'NUMBER' - UTILITY TO

NUMBER OR RE-NUMBER BASIC

PROGRAMS

PROGRAM DESCRIPTION

NUMBER is a FORTRAN program that reads a BASIC program and either numbers or re-numbers its statements.

NUMBER is invoked as an external command by typing:

NUMBER

The program, NUMBER, then responds:

PARAMETERS

The parameters that may be specified are:

IFILE - input file name (first 6 characters)
OFILE - output file name (first 6 characters)
START - starting statement number (Decimal $1 \leq$ START ≤ 9999)
INCR - statement number increment (Decimal $1 \leq$ INCR ≤ 9999)

The parameters, OFILE, START, and INCR are optional. However, if INCR is specified, START must be specified also. If OFILE is omitted, the output is placed in IFILE. If START and INCR are both omitted, their value is 1. If INCR alone is omitted, its value is 1.

The input file specified by IFILE can be either a completely numbered file or a partially numbered file. If every statement has a statement number, the file is re-numbered in the order of statement numbers of the input file. For example, if the input file contained the following statements:

```
 10 DIM A(9,9)
 12 MAT A= ZER
 30 W1=0
 35 W2=0
 50 FOR K=1 TO 7
 60 FOR I=1 TO 3
 70 FOR J=I-1 TO 3
 80 IF J<=1 THEN 95
 90 GOSUB 290
95 NEXT J
110 NEXT I
120 NEXT K
130 MATPRINT A
139 STOP
```

and the following sequence of command lines is initiated

```
NUMBER                = user types

PARAMETERS            = system responds

INMAT OUTMAT 10 10    = user types
```

where INMAT is the input file, OUTMAT is the output file, the starting statement number is 10, and increment is 10. The output becomes:

```
 10 DIM A(9,9)
 20 MAT A= ZER
 30 W1=0
 40 W2=0
 50 FOR K=1 TO 7
 60 FOR I=1 TO 3
 70 FOR J=I-1 TO 3
 80 IF J<=1 THEN 100
 90 GOSUB 290
100 NEXT J
110 NEXT I
120 NEXT K
130 MATPRINT A
140 STOP
```

The input file specified by IFILE can be only partially numbered, NUMBER numbers statements in this type of file in the order of their occurrence. In the following example, the file is sequential and only the referenced lines contain numbers.

```
REM TEST OF THE NUMBER PROGRAM.
REM OBJECT IS TO SEE HOW A PARTIALLY NUMBERED PROGRAM IS HANDLED.
REM
10 INPUT N
PRINT 'IS THIS THE END':
INPUT N$
IF N$ ='END' THEN 99
GOSUB 500
GO TO 10
99 PRINT 'THIS IS THE LIVING END.'
END
REM *****BEGIN SUBROUTINES.*****
500 LET X = N  ^  2
Y = 2*N
S = N/2
Q = SQR(N)
PRINT N:'***',Q,S,Y,X
PRINT
599 RETURN
```

The results of the interaction

```
NUMBER
PARAMETERS
PTESTN  2  2
```

are as follows:

```
 2  REM TEST OF THE NUMBER PROGRAM.
 4  REM OBJECT IS TO SEE HOW A PARTIALLY NUMBERED PROGRAM IS HANDLED.
 6  REM
 8  INPUT N
10  PRINT 'IS THIS THE END':
12  INPUT N$
14  IF N$='END' THEN 20
16  GOSUB 26
18  GOTO 8
20  PRINT 'THIS IS THE LIVING END.'
22  END
24  REM *****BEGIN SUBROUTINES.*****
26  LET X=N^2
28  Y=2*N
30  S=N/2
32  Q=SQR(N)
34  PRINT N:'***',Q,S,Y,X
36  PRINT
38  RETURN
```

Note that statements are numbered by NUMBER using only as many digits as required. Thus, '599' in the above example becomes '38'.

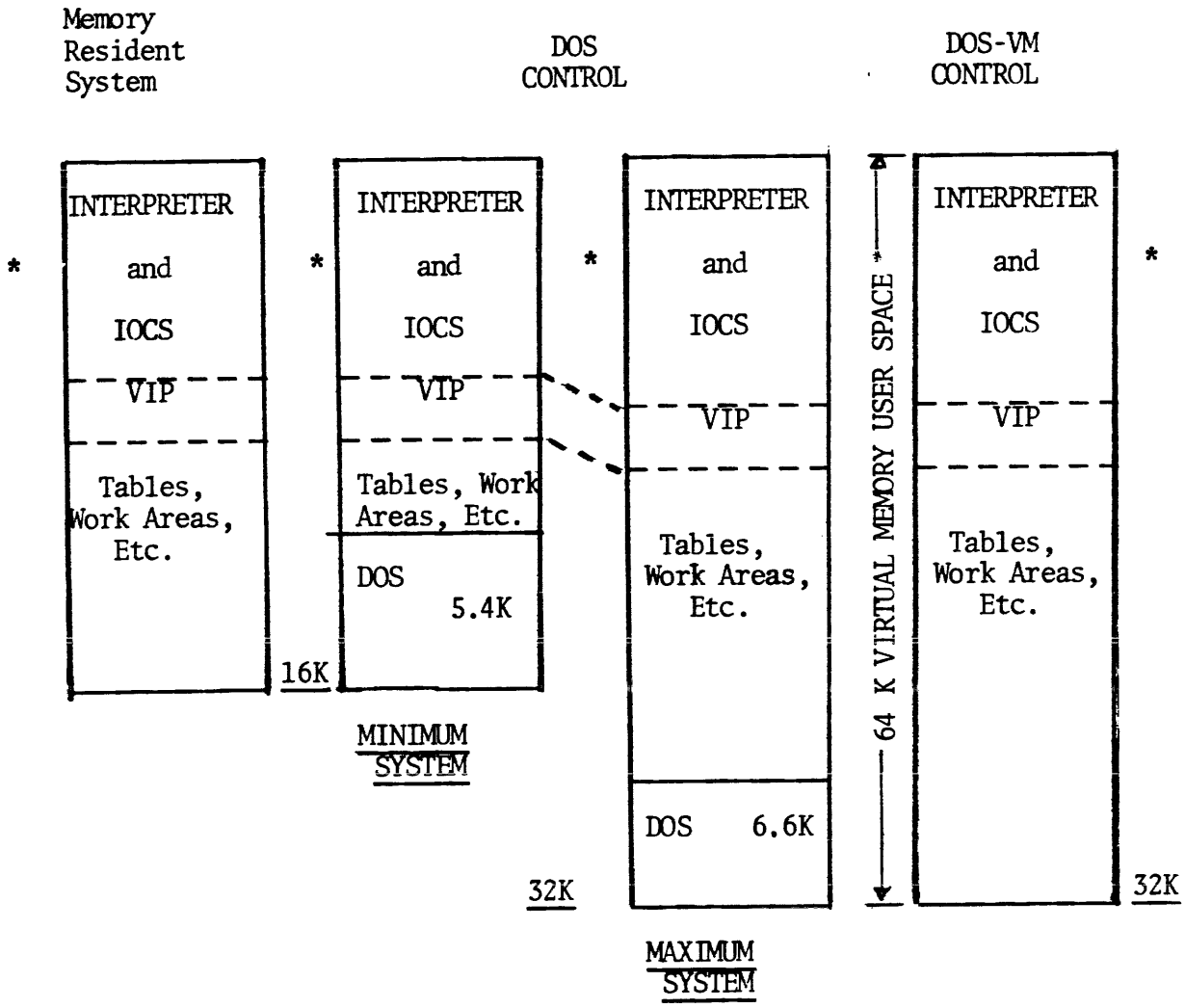
When the NUMBER program completes execution, the input file, IFILE, is closed; and the output file, OFILE, is also closed if it was opened.

ERROR MESSAGES

Messages	Remarks
BAD PARAMETERS	- If either START or INCR are specified with more than 4 digits, NUMBER requests a new parameter line.
XXXXXX NOT FOUND	- The specified IFILE does not exist. Control returns to DOS.
XXXX DUP LINE NUMBER	- XXXX occurs as a line number more than once. Control returns to DOS.
INPUT FILE NULL	- The specified IFILE is empty. Control returns to DOS.
MEMORY OVERFLOW	- There is not enough memory to contain a map of line numbers. Control returns to DOS.
LINE NUMBER OVERFLOW	- A new line number 9999. Control returns to DOS.

APPENDIX E

MEMORY REQUIREMENTS



*See Tables on Page E-2 for memory allocation

	Single Precision BASIC	Double Precision BASIC
BASIC	7.2K	**
BASIC with PRINT USING	8.0K	**
BASIC with MATRIX	8.2K	**
BASIC with PRINT USING & MATRIX	10.2K	**

Interpreter and IOCS Memory Allocation

High Speed and Floating Point Arithmetic	Memory Required Single-Prec.	Memory Required Double-Prec.
Neither	850 wds.	**
High Speed Arith. only	640 wds.	**
High Speed & Floating Point	0 wds.	**

VIP (Virtual Instruction Package)

Memory Requirements

** to be supplied when double precision is available.

Function	Memory Required
Fixed Table	700 words (single prec.) 1300 words (Double prec.)
Program Storage	APPROX. 1 Word/2 char.
STATEMENT	2 Words/Statement Index
Packet Storage	String values and FOR-NEXT loop parameters
Array Storage	Dependent upon size.

Memory Allocations for
Tables, Work Areas, etc.

INDEX