HONEYWELL

# LEVEL 68 MULTICS BASIC MANUAL

SOFTWARE

LEVEL 68

# MULTICS BASIC MANUAL

SUBJECT

General Description, Capabilities, Rules and Definitions, User Interfaces, Statements, and Input/Output of the BASIC Language on the Multics System

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

AM82-01

February 1981

Honeywell

# PREFACE

This reference manual completely describes the BASIC language on the Multics system. It does not describe the BASIC compiler. For information on the BASIC compiler, the reader is referred to the basic command description in the MULTICS PROGRAMMERS' MANUAL, COMMANDS AND ACTIVE FUNCTIONS. Also, this manual does not attempt to provide the reader with extensive knowledge of the Multics system. The reader is referred to the Multics Programmers' Manual Introduction for an introduction to Multics use and to the other volumes of the Multics Programmers' Manual (MPM) for a thorough discussion of the Multics system. The complete MPM consists of seven manuals although only four are referenced in this manual. They are as follows:

| Document | Referred to in Text As |
|---|---|
| Introduction (Order No. AG90) | MPM Introduction |
| Reference Guide (Order No. AG91) | MPM Reference Guide |
| Command and Active Functions (Order No. AG92) | MPM Commands |
| Subroutines (Order No. AG93) | MPM Subroutines |

The MPM Reference Guide contains general information about the Multics command and programming environments.

The MPM Commands gives the syntax and a complete description of selected standard Multics system commands and active functions.

File No.: 1L13

The MPM Subroutines contain descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each and a description of the I/O modules. In addition to the MPM, the reader is referred to the Multics FAST Subsystem Reference Guide (Order No. AU25-01) describing the time-sharing facility supporting BASIC and FORTRAN program development.

This is the first revision of the BASIC manual. Features new to the manual are:

extended precision
let keyword is now optional
multiple statements per line

CONTENTS

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

CONTENTS (cont)

# SECTION 1

# INTRODUCTION

## FORMAT OF STATEMENTS

A BASIC program is a sequence of numbered statements most of which are identified by a keyword. The source program text consists of a Multics segment containing ASCII characters divided into lines by "newline" characters (the ASCII character whose octal code is 12). Each line of the source program contains one or more BASIC statements. Blank lines are allowed. Multiple statements can appear on one line but must be separated by a backslash (\) character. A statement that spans several lines is not allowed.

The following statements constitute a complete BASIC program, it computes and prints the sum and difference of two numbers specified by the user when the program is executed.

```
100 input x,y
200 print x+y, x-y
300 end
or:
100 input x,y\print x+y, x-y
200 end
```

### Line Numbers

The line or statement number is an unsigned decimal integer greater than or equal to 0 and less than or equal to 99999 that is used to label the statement. The line number must begin in the first position of the source line; the line number field is terminated by the first nondigit in the line.

Each line in the source program must have a line number that is greater than the number of the preceding line. Other commands (such as fast) that make use of the BASIC compiler can use line numbers to control editing of the source program; if so, the maximum value of a line number may be restricted to a lower value than that imposed by BASIC.

### Keywords

The statement keyword is an English word that immediately follows the line number or backslash and serves to identify the type of statement. The interpretation of the characters that follow the keyword depends on the type of statement. Some examples of BASIC keywords are:

```
let
print
if
rem
```

### Character Processing

The BASIC compiler ignores blanks and tab characters and converts uppercase characters to lowercase ones except where they occur within quoted strings. Thus the following statements are all equivalent:

```
100 GOTO485
100 goto 485
100 go TO 4 8 5
```

The length of the line after blanks and tab characters have been removed is limited to 256 characters.

## ORDER OF EXECUTION

The statement in a program with the lowest line number is the first statement to be executed. Unless one of the control statements is executed, statements are executed sequentially according to line number. Execution of the program ceases if an end statement or a stop statement is executed.

## REMARKS

The BASIC compiler normally looks at all the characters in a statement. BASIC provides two means by which the user can indicate that a sequence of characters is to be ignored by the compiler: the remark statement and apostrophes.

## Remark Statement

The remark statement consists of the keyword "rem" followed by an arbitrary sequence of characters that can contain any ASCII characters except newline or backslash characters. The compiler ignores the rem statement; it has no effect when encountered during execution. An example of a rem statement is:

```
173 rem Test for Convergence
```

A statement consisting of only a line number is treated as if it were a rem statement.

## Apostrophes

When the BASIC compiler finds an apostrophe or an acute accent (the ASCII character whose octal code is 47) outside of a string constant, it ignores the apostrophe and all characters between it and either the newline character that terminates the line or the backslash character that terminates the statement, whichever comes first. This allows comments to be written on the same line as a BASIC statement. If the apostrophe immediately follows the line number, the line is treated as a rem statement. The following lines show how the apostrophe can be used.

```
800 goto 200 ' repeat for next value
829 ' find total cost
```

## BASIC PROGRAM STRUCTURE

A BASIC source file may consist of a main program, one or more subroutines, or a main program followed by one or more subroutines, each having its own independent set of variables. If the first executable statement of a program does not begin with the keyword sub, it is the main program whose body is terminated by the end statement. The name of a main program is the same as the primary name of the source program; for example, the main program corresponding to the source file named alpha.basic would be alpha.

A main program can be directly invoked by the BASIC compiler as part of the compile-and-go mode of execution; it can be invoked by the Multics command processor if a Multics object segment exists for the program; or it can be called as a subroutine by a separately compiled BASIC, PL/I, FORTRAN, or ALM program.

A subroutine immediately follows the optional main program. It has its own set of variables. The name of a subroutine and the arguments it takes are specified by a sub statement. The name of a subroutine is given as an ASCII character string of 32 characters or less. The body of the subroutine is delimited by a sub statement at the start of the subroutine and a subend statement at the end. It is not possible to nest subroutine definitions.

A subroutine is normally called by means of a call statement. As with main programs, a subroutine can be called from a BASIC, PL/I, FORTRAN, or ALM program. A subroutine that has no arguments can be called by the Multics command processor.

In the rest of the manual, the word program is used to mean either main program or subroutine, whether or not they are compiled together.

## Allocation of Storage

Storage is allocated for all the variables in a program when control enters the program and is freed when control leaves the program. The variables of a program are accessible only when the program is active; the values of all variables are lost when the program returns.

If a program is called recursively (called while a previous call is still active), the storage allocated for the variables of the previous use of the program is "pushed down" at each entrance to the program and "popped up" at each return. The variables associated with a previous invocation are not accessible to a subsequent activation of a program. There is no means, other than the use of files, by which the value of a variable can be preserved between invocations of a program.

## Writing and Compiling a BASIC Program

A program can be written online using any one of the Multics editors such as qedx for hard-copy terminals or emacs for video terminals. For detailed information on the qedx editor, see the Qedx Text Editor User's Guide, Order No. CG40, or for emacs, see the Emacs Text Editor User's Guide, Order No. CH27. The source program may consist of the source for one or more subroutines, a single main program, or a main program followed by one or more subroutines.

Once the program is input and edited, it is compiled by the BASIC compiler which translates the program and then either executes a temporary object segment or produces a standard Multics object segment. The compiler is invoked by the basic command discussed in detail in the MPM Commands.

## BASIC SEARCH MECHANISM

When a BASIC call statement is executed, BASIC first searches the caller's object segment for a subroutine with the specified name. If none is found, BASIC looks for the program in the storage system. In this case, the format of the name determines the search methods.

The subroutine name used in a BASIC call statement can be a segment name, e.g., alpha; a segment name and entry name pair, e.g., alpha$beta; or either of these preceded by an arbitrary pathname, e.g., >udd>Person_id>Project_id>alpha. When a pathname is not present, BASIC uses the normal Multics search algorithm to locate the specified segment. If a pathname is given, BASIC attempts to initiate the specified segment using the specified segment name. If no other segment is known by this name, the name/number association is established and the segment that was found is used for the call statement. If the specified segment is not found or if some other segment is already known by the specified name, the call statement is in error and execution of the BASIC program is terminated.

## SAMPLE PROGRAM

The following BASIC program consists of one main program and one subroutine. The main program reads input from the terminal; calls the subroutine, which computes the integral of the defined functions; and then prints the difference between the results returned by the subroutine.

```
100  input l,u
110  def fna(x) = 1 + x*x
120  def fnb(x) = 1 - x*x
130  call "integrate":fna,l,u,z1
140  call "integrate":fnb,l,u,z2
150  print z1-z2
160  end
210  sub "integrate":fnc,l,u,z
220
  .
  .  (body of program omitted)
  .
270  subend
```

## SECTION 2

## TYPES OF DATA

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

### NUMERIC ARGUMENTS

A numeric value is a floating-point number that is single precision by default. The mantissa of a numeric value can represent approximately eight decimal digits of precision; the exponent can be between -39 and +38. Integers whose magnitude is less than 134,217,728 are represented exactly.

If the program is compiled in extended precision mode (see Appendix D), all numeric values are double precision floating point numbers. In this case, the mantissa can represent approximately 19 decimal digits of precision. The exponent can be between -39 and 38. Integers whose magnitude is less than 2**63(9223372036854775807) are represented exactly.

A numeric constant is written as a signed decimal number that can contain a decimal point followed by an exponent. The exponent field, which is optional, is written as the letter "e" (or the letter "E" since BASIC converts uppercase characters to lowercase characters) followed by an optionally signed integer constant. If the decimal point is omitted, it is assumed to be immediately to the right of the rightmost digit. If the sign of either the decimal number or the exponent is omitted, a plus is assumed.

Examples:

```
0
14
1.5
-7.0e3
a+5.12 e - 5
-.333E-10
5 e 5
```

## STRING VALUES

A string value is a string of from 0 to 4095 ASCII characters, inclusive.

A string constant is written as a set of 0 or more contiguous ASCII characters enclosed in quotation marks. A quotation mark can be included in the string by immediately preceding it by another quotation mark. A string constant can contain any ASCII character except a newline character; a string constant must fit entirely in one BASIC statement.

Examples:

```
""""
    (Result: ")
"This is a character string constant"
    (Result: This is a character string constant)
"She said, ""I love you!"""
    (Result: She said, "I love you!")
```

## SCALAR VARIABLES

A scalar variable is implicitly declared through its appearance in the source program. The type of a scalar variable is completely determined by the spelling of the variable name.

### Numeric Scalars

The name of a scalar numeric variable is either a letter or a letter followed by a digit. Numeric variables are initialized to 0 at the start of the program in which they are defined.

Examples:

```
a
C3
```

String Scalars

The name of a scalar string variable is either a single letter followed by a dollar sign or a single letter followed by a digit followed by a dollar sign. String variables are initialized to the zero-length string at the start of the program in which the variable is defined. The length of a string variable is automatically set to the length of the string being assigned to it.

Examples:

```
a$
F3$
```

ARRAY VARIABLES

An array is an ordered set of values, all of which have the same type. The elements of an array are stored in contiguous storage in row-major order (rightmost subscript varies most rapidly). An array can have either one dimension, in which case it is called a vector, or two dimensions, in which case it is called a matrix. The first subscript of a matrix selects a row of the matrix; the second subscript selects a column.

Array Declarations

An array can be explictly declared in a dim statement or can be implicitly declared by its use in the source program. If the first use of the array name is in a dim statement, the number of dimensions and the number of elements are given in the dim statement.

If the first use of an array name is in an executable statement, the number of dimensions in the array is determined by the context in which the array name is found. The following contexts declare the array as a vector:

- use with one subscript
- use in a call statement where a vector is required
- use in a change statement
- use in an array input/output statement without redimensioning

The following contexts declare the array as a matrix:

- use with two subscripts
- use in a call statement where a matrix is required

Use of an array in a mat-assign statement can cause the array to get declared either as a vector or as a matrix, depending on the other arrays used in the statement.

The lower bound of an array dimension is always 0. The upper bound depends on the manner in which the array was declared. If an array was explicitly declared, the number of elements in each dimension is given in the dim statement. If an array is implicitly declared, BASIC uses an upper bound of 10 for each dimension; thus 11 elements are reserved for a vector and 121 elements are reserved for a matrix.

## Array Bounds

An array really has two sets of bounds: the original bounds and the current bounds.

The original bounds of an array are established by the dim statement in which the array is declared or by the implicit dimension rules previously specified. The original bounds of an array specify the total amount of storage allocated for the array and do not change during execution of the BASIC program.

The current bounds of an array are initially the same as the original bounds. The current bounds indicate the total amount of storage currently being used for array elements; the current bounds (but not the number of dimensions) can be changed by the process known as redimensioning described in Section 6. The amount of storage specified by the current bounds must always be less than or equal to the amount of storage specified by the original bounds.

## Array Element References

A reference to an array element consists of the array name followed by a parenthesized list of subscripts.

Syntax:

```
    b(s1)    or    b(s1,s2)
```

where b is the array name and s1 and s2 are numeric expressions. The value of a subscript expression is truncated to an integer value before it is used to locate the specified array element. The truncated value of the subscript expression must be within the range of the corresponding array dimension. The number of subscripts in an array element reference must be the same as the number of array bounds.

Examples:

```
a(5)
b(i+j,3*k-2)
c(a(j)+4)
```

## Numeric Arrays

The name of a numeric array is a single letter. All the elements of the array are initialized to 0 at the start of the program in which the array is defined.

## String Arrays

The name of a string array is a single letter followed by a dollar sign. The elements of a string array can all have different lengths. All the elements of the array are initialized to the zero-length string at the start of the program in which the array is defined.

Examples:

```
a$(5)
b$(i+j,3*k-2)
c$(a(j)/4)
```

## RELATIONSHIP OF NAMES

The same name can be used for as many as four different variables in a BASIC program; the context in which the name is used identifies the scalar or array intended. Thus, in a single program, a user can have a scalar numeric variable b, a scalar string variable b$, a numeric array b, and a string array b$, all of which are distinct variables.

The following expression illustrates all four uses of the name b:

```
str$(b(val(b$))) & b$(b)
```

## REFERENCES

Throughout this document, the word "reference" means a reference to a scalar variable or a reference to a subscripted array element. Whenever a particular type of reference is required, the terms "numeric reference" and "string reference" are used. An "array reference" is a reference to an array used without subscripts.

## LISTS

Throughout this document, the word "list" means a sequence of one or more items separated by commas or some other specified separator. A list cannot contain two consecutive elements or two consecutive separators. The following are examples of lists of integers:

```
1
1,2
1,2,3
```

# SECTION 3

# EXPRESSIONS

BASIC expressions are constructed from operators and operands. An operand can consist of a constant, a scalar variable or subscripted array element, a function reference, or the result of another operator. Operators that require two operands are called binary operators, and 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 the string operator; string operands must not be used with the numeric operator. There is no implicit conversion between numeric and string values; explicit conversion functions must be used to convert from one data type to the other.

Throughout this document, the word "expression" means an arbitrarily complicated expression that can range from a single constant to a complicated construct containing many operators and parentheses. When a particular type of expression is intended, the terms "numeric expression" and "string expression" are used.

## NUMERIC EXPRESSIONS

BASIC defines seven operators that operate on numeric operands to produce a numeric value:

| Operator | Meaning | Example |
|---|---|---|
| + | plus | + a |
| - | minus | - a |
| + | addition | a + b |
| - | subtraction | a - b |
| * | multiplication | a * b |
| / | division | a / b |
| ^ | exponentiation | a ^ b |

The operators have their normal arithmetic meaning. The operations are performed using the floating-point instruction set of the Multics machine. Addition, subtraction, multiplication, and exponentiation of integer values are exact, provided the magnitudes of the operands and result are less than 2^27 (134,217,728) for single precision or 2^63 for extended precision.

The order in which these operators are evaluated is determined by special rules of precedence. The precedence of the numeric operators is:

| Precedence | Operator |
|---|---|
| 4 (highest) | unary -, unary + |
| 3 | ^ |
| 2 | * / |
| 1 (lowest) | + - |

Operators with higher precedence are evaluated first. Operators of equal precedence are evaluated from left to right, except for the exponentiation and unary + and - operators, which are evaluated from right to left. For example, the expression

```
a + b + c * d * e ^ f ^ g
```

is interpreted as

```
(a + b) + ((c * d) * (e ^ (f ^ g)))
```

Parentheses can also be used to control the order of expression evaluation.

Examples:

```
a + b/c
(a + b)/c
(a - b7 * 3.1415)/(c1 + d^2)
a(i,j) + b(j-1,i+5)
-a * b
```

## STRING EXPRESSIONS

String expressions in BASIC are constructed using the concatenation operator &. This operator combines two string values to produce a string whose value is the characters in the first string immediately followed by the characters in the second string.

Examples:

```
"hello " & "there"  (Result: "hello there")
"upper" & "case"    (Result: "uppercase")
a$ & b$ & c$
a$(i) & a$(i+1)
```

## FUNCTIONS

A function reference consists of a BASIC function name or a user-defined function name optionally followed by a parenthesized argument list containing one or more arguments. The arguments used in a function invocation must match the number and type of arguments expected by the function. No conversion is done to match the argument provided with the argument expected. Function references are evaluated at the point where their value is required and do not affect the order of operator evaluation. All function arguments are evaluated before the function is evaluated.

### BASIC Functions

BASIC provides a variety of functions for computing commonly used functions and for interrogating the operating environment of the program. Numeric function names consist of three letters; string function names consist of three letters followed by a dollar sign. Except where explicitly stated otherwise, a numeric argument of a function can be any arbitrarily complicated numeric expression, and a string argument of a function can be any arbitrarily complicated string expression.

The following list gives the numeric and string functions provided by BASIC; functions related to files are listed in Section 4. In all of the descriptions that follow, x indicates an arbitrary numeric expression, i and j indicate arbitrary numeric expressions that are truncated to yield an integer value, and a$ and b$ indicate arbitrary string expressions.

| Function | Description |
|---|---|
| abs(x) | The absolute value of x. |
| asc(c) | The decimal number corresponding to the single ASCII character or two- or three-letter character abbreviation c. Any single ASCII character can appear except quote, newline, apostrophe, space, and tab; character abbreviations are listed in Appendix A. |
| arg$(n) | The value of the nth command argument supplied by the Multics command processor. This function can be used when a BASIC main program has been called as a Multics command. |
| atn(x) | The arctangent of x in radians (i.e., the angle whose tangent is x), where the angle is in the range -pi/2 to +pi/2. |
| chr$(x) | The one-character string that consists of the ASCII character with numeric code mod(int(x),128). (See Appendix A.) |
| clg(x) | The logarithm of x to the base 10. |
| clk$ | An eight-character string that gives the time of day in the form HH:MM:SS. |
| cnt | The number of arguments supplied by the Multics command processor. This function can be used when a BASIC main program has been called as a Multics command. |
| cos(x) | The cosine of x, where x is in radians. |
| cot(x) | The cotangent of x, where x is in radians. |
| dat$ | An eight-character string that gives the current date in the form MM/DD/YY. |
| det | The determinant of the last matrix that was inverted in this program using the matrix function inv. (See Section 6.) |
| exp(x) | The exponential of x (i.e., the value of e raised to the power x). |
| int(x) | The largest integer not greater than x. |
| len(a$) | The number of characters in the string a$. |
| log(x) | The logarithm of x to the base e. |

| Function | Description |
|---|---|
| max(x1,...,xn) | The maximum of n numeric values. This function allows an arbitrary number of arguments. |
| min(x1,...,xn) | The minimum of n numeric values. This function allows an arbitrary number of arguments. |
| mod(x,y) | The modulus function x - y * int(x/y); the value x is returned if y is 0. |
| num | The number of data items transmitted into the last array by a mat-input statement. (See Section 6.) |
| pos(a$,b$,i) | The location in string a$ of the first occurrence of string b$, starting at or after position i in a$. A 0 is returned if string b$ does not occur in the specified substring of a$ or if i <1 or i> len(a$). |
| rnd | The next pseudorandom number in a sequence of uniformly distributed pseudorandom numbers greater than or equal to 0 and less than 1. The period of the sequence is 2^35 - 1. |
| seg$(a$,i,j) | The substring of a$ that consists of the characters between positions i' and j' inclusive, where i' = max(i,1) and j' = min(j,len(a$)). A zero-length string is returned if j' < i'; otherwise, the length is j'-i'+1. |
| sgn(x) | The signum of x: -1 if x < 0, 0 if x = 0, and +1 if x > 0. |
| sin(x) | The sine of x, where x is in radians. |
| sqr(x) | The positive square root of x. |
| sst$(a$,i,j) | The substring of a$ that consists of j' characters starting at the character in position i', where i' = max(i,1) and j' = max(min(j,len(a$)-i'+1),0). |
| str$(x) | The string that is the decimal representation of the numeric value of x. The conversion follows the rules for printed output. (See Section 5.) |

| Function | Description |
|---|---|
| tan(x) | The tangent of x, where x is in radians. |
| tim | The elapsed running time of the program in seconds. This value is determined from the microsecond clock used by the Multics system. |
| tst(a$) | This function returns a value of 1 if the string a$ can successfully be converted to a numeric value according to the rules for numeric input; 0 is returned if the string a$ does not represent a valid numeric constant. |
| usr$ | A string giving the name of the user (e.g., Jones). |
| val(a$) | The value of the number whose decimal representation is a$. |

## User Functions

In addition to the standard functions that it provides, BASIC allows the user to define his own functions. These function definitions are local to the program in which they appear. Two forms of function definition are permitted: single line functions and multiple line functions.

A single line function returns the value of a numeric or string expression that can depend on the parameters, if any, of the function. A multiple line function can perform more complicated computations before it returns its result.

The name of a user-defined numeric function consists of the letters "fn" followed by a single letter. The name of a user-defined string function consists of the letters "fn" followed by a single letter followed by a dollar sign. The same letter can be used for both a string function and a numeric function in the same program.

Examples:

```
fna
fna$
```

A reference to a user-defined function consists of the name of the function optionally followed by a parenthesized argument list containing one or more arguments. The arguments supplied in a reference to a user-defined function must agree in number and type with the parameters expected by the function; no conversion is done to match the argument provided with the parameter expected. Arguments are passed to a user-defined function "by value"; this allows the function to assign a value to a parameter without changing the corresponding argument.

Multiple line functions can be defined with local variables. A variable used in a function body that is not a parameter or a local variable of the function is said to be a global variable. A global variable is defined in the program that contains the function definition.

A multiple line function can call itself recursively, i.e., the function can be invoked while one or more previous invocations are still active. The recursive invocation can be direct, as the result of a use of the function from within its own definition, or indirect, as the result of a call from some other function.

## SECTION 4

## FILES

A BASIC file is a set of data external to the BASIC program. A file either is associated with an input/output device, or it resides in the Multics storage system as one or more segments. The data in a Multics BASIC file are organized into sequential records. The contents of a file are made available to the BASIC program by the execution of input/output statements that transmit data between the file and the program.

BASIC permits two classes of files: terminal format files and random access files.

### TERMINAL FORMAT FILES

A terminal format file is a string of ASCII characters organized into lines ending with newline characters. A line in a terminal format file consists of the contiguous string of characters between a newline character and the next newline character in the file. The length of a line of a terminal format file can be from 0 to 4095 characters, inclusive.

A terminal format file is read and written through a Multics I/O switch that is attached to the user's terminal, or to a file that resides in the Multics storage system, or to a specific input/output device. Terminal format files are accessed in a sequential manner; it is not possible to access an arbitrary line in a terminal format file. An I/O switch serves as a channel through which input/output is performed.

## RANDOM ACCESS FILES

A random access file is a collection of data that can be accessed in a nonsequential manner. Input/output operations on random files normally process the file sequentially, but operations are available that permit the user to access any arbitrary datum. Random access files are not read or written through a Multics I/O switch. Each random access file resides in a single segment within the Multics storage system.

A random access file is called a random numeric file or a random string file according to the type of data it contains. A random access file can contain only a single type of data; numeric data cannot be stored in a string file, and string data cannot be stored in a numeric file.

Numeric files used by programs compiled in extended precision mode must have double precision values. Likewise, numeric files used by single precision programs must have single precision values. (See Appendix D for information about converting files.)

### Random Numeric Files

A random numeric file can be thought of as a vector of numeric values. Each record in a random numeric file consists of a single numeric datum.

### Random String Files

A random string file can be thought of as a vector of strings of some previously specified maximum length; the default value of this maximum length is 10 characters, the maximum length can be as large as 4095 characters. Each record in a random string file consists of a single string datum.

When a string value shorter than the maximum length is written into a string file, it is not extended to the maximum length. The current length of the string value is stored in the file along with the characters that comprise the string value. Writing a string into a string file and then reading it back results in a string of identical contents and length.

## FILE NAMES

The name of a BASIC file is a string of ASCII characters. This string is used by the BASIC runtime system to locate the file. Whenever a file name is required by the BASIC program, the user can write an arbitrary string expression. There are two kinds of file names: those that have a colon as the first character and those that do not.

When a colon is the first character of a file name, the file name specifies a Multics I/O switch name. An I/O switch serves as a channel through which input/output is performed. By specifying a switch, rather than a specific device or file, a BASIC program becomes device or file independent. The switch can be attached to a different device or file each time the program is executed. A file name of the form:

```
:name
```

connects the BASIC file to the I/O switch name, which must already be properly attached. A file name of the form:

```
:name attach-description
```

connects the BASIC file to the I/O switch name; attach-description specifies the manner in which the switch should be attached if not already attached. The types of attachments that can be made are described in Appendix C.

If BASIC attaches the switch, it also opens, positions, closes, and detaches the switch at the termination of the BASIC program. If the switch is already attached, BASIC opens, positions, and closes it but does not detach it. Finally, if the file name specifies an I/O switch that is both attached and open, BASIC does not position, close, or detach the switch.

File names that begin with a colon cannot be used for random access files. Examples of file names that have a colon as the first character are:

```
:error_output
:xxx vfile_ xxx_file
:input record_stream_ -target ntape_ 123abc,9track -raw
```

A file name that does not begin with a colon is interpreted as a Multics pathname that specifies a segment in the Multics storage system. The pathname can be either absolute or relative. (Refer to the New Users' Introduction to Multics Part I, Order No. CH24 for a description of absolute and relative pathnames.) This kind of file name must satisfy all constraints on pathnames (refer to the MPM Reference Guide) that are enforced by the Multics operating system. Examples of this type of file name are:

```
error_output
data
>udd>projectid>personid>filea
<input
```

## FILE NUMBERS

A BASIC program refers to its files by means of a file number. A BASIC file number is an integer from 0 to 16, inclusive. File number 0 always refers to the user's terminal, which is treated as a terminal format file.

The correspondence between a file number and a file name is established by the file statement. A file is called "open" if it is currently assigned a file number and is called "closed" otherwise.

A file statement results in an attempt to locate the specified file, either as an I/O attachment or as a Multics segment. If the file is located, the BASIC runtime system determines the type and attaches the file appropriately. Errors that can be detected include: an invalid file number, an invalid file name, no read access, a type not used by BASIC programs, and a numeric file that has a precision different from the program. If the file is not located, it will be created when first used. If an I/O attachment is specified, there must be a valid attach description if the file is not already attached, and if the file is already open, it must be for stream input or stream output.

A file remains open until it is closed. A file can be closed in one of two ways:

1. When control returns from a BASIC program, either normally or abnormally, all files opened by the program are automatically closed.

2. A file is closed if its file number is used in a subsequent file statement in the same program.

## FILE EXPRESSIONS

Whenever a file number is required in a BASIC program, the user can write an arbitrary numeric expression whose value is truncated to an integer before it is used. Throughout this document the term "file expression" signifies a numeric expression that results in an integer value from 0 to 16, inclusive.

## TEMPORARY FILES

The file name "*" refers to a temporary file that is created by the file statement that opens it. A temporary file is deleted at the termination of the program that created it. Each use of the file name "*" in a file statement results in the creation of a new file that is distinct from any other temporary files previously created.

## FILE ATTRIBUTES

Each BASIC file has associated with it a type, a length, a margin, and a pointer.

### File Type

A BASIC file has one of three types: terminal format, random numeric, or random string. The type of a nonempty BASIC file is uniquely determined by its contents. The type of an empty file is set by the first output to the file: a print statement causes the file to become terminal format; a write statement causes the file to become random numeric or random string depending on the type of the first datum written. The type of a BASIC file can be changed if and only if the file is empty.

### File Length

The length of a terminal format file is the number of lines in the file. The length of a random file is the number of data elements in the file. The length of an empty file is 0. The length of a file can be changed by print, scratch, or write statements.

### File Margin

The margin of a terminal format file is the maximum number of characters that can be placed in a line of output before the BASIC runtime system automatically generates a newline character and starts a new line of output. Each character placed in the line is treated as if it advanced the terminal print head by one position; nonprinting characters such as tab or backspace are not treated specially. A margin of 0 for a terminal format file means that the output line is to be treated as if it were infinitely long; in this case, the line overflow check is suppressed. The default margin setting for a terminal format file is 75 characters.

The margin of a random string file is the length of the largest string that can be written into the file without being truncated; the default margin value is 10 characters. The margin of a random numeric file has no effect on the contents of the file but must, by convention, be 1.

The margin of any BASIC file must be from 0 to 4095, inclusive. The margin of a file can be changed by the margin statement.

### File Pointer

The file pointer specifies the next location in a file that is affected by an input/output operation on the file. When a file is first opened, the file pointer points to the beginning of the file.

The first print operation on a terminal format file causes the file pointer to be set to the end of the file before any characters are transmitted to the file. The file pointer of a terminal format file is changed by the input, linput, and print statements. The reset statement can be used to position the file pointer to the beginning of the file.

The file pointer of a random file that contains N values can range from 0 to N-1. The file pointer of a random file is changed by the read and write statements. The file pointer can be changed to point to any value in the file by means of the reset statement.

## FUNCTIONS

The following functions are provided for obtaining information related to the status of BASIC files; n stands for a file expression, while a$ indicates an arbitrary string expression.

| Function | Description |
|---|---|
| hps(#n) | The current value of the horizontal print position (number of characters in partial line) of the terminal format file assigned file number n. |
| loc(#n) | The current value of the file pointer for the random access file assigned file number n; 0 is returned if the file is empty. |
| lof(#n) | The current length (number of data items) of the random access file assigned file number n; 0 is returned if the file is empty. |

| Function | Description |
|---|---|
| mar(#n) | The current margin of the file assigned file number n. |
| per(#n,a$) | The value +1 if the operation specified by a$ is permitted for file number n, 0 if the operation is not permitted, and -1 if a$ does not specify one of the operations input, linput, print, read, reset, scratch, or write. An operation is not permitted if the type of the file is incorrect or if there is no write access in the case of output operations. |
| typ(#n,a$) | The value +1 if file number n is of type a$, 0 if file number n is not of type a$, and -1 if a$ does not specify one of the types numeric, string, terminal, tty, or any. Any open file has type any. An empty file has any type except tty. |

## NOTES

In Multics the # is a special character and in order for it not to perform its delete function it must be preceded by a backslash (\). See the MPM Communications I/O, Order No. CC92, for further information on special characters.

# SECTION 5

# STATEMENTS

This section discusses all BASIC statements except the array manipulation statements described in Section 6. When used in a program, each statement must have a line number. For the sake of simplicity, line numbers are omitted here.

## CALL STATEMENT

Syntax:

```
    call s$
       or
    call s$ : list
```

where s$ is a string expression, and list is a list of argument specifications. An argument specification can be an expression, an array argument, a function argument, or a file argument.

Semantics:

The call statement transfers control to the program whose name is given by the value of the string expression s$. The program can be compiled with the program in which the call statement is located or it can be separately compiled, in which case the called program can be written in some other language. When the called program returns (in the case of a BASIC program by executing a stop, subend, or end statement) execution continues with the statement following the call statement.

The program name can be a simple segment name, a combined segment and entry name in the form segment$entry, or either of these prefixed by an arbitrary pathname. If the specified program is not found in the current program, the Multics search rules in effect at the time of the call are used to find it or, if a path is specified, the indicated segment is initiated. Some examples of program names are:

```
"integrate"
"solve$initialize"
"solve"
">udd>projectid>personid>library>ROUTINE"
a$ & b$
```

## Arguments

The number and type of arguments passed to a BASIC subroutine must agree in number and type with the parameters expected by the subroutine. Four types of arguments can be passed: expressions, arrays, functions, and files.

### EXPRESSION ARGUMENTS

Expression arguments are passed to the called subroutine by location. If an argument in the call statement is a scalar variable or a reference to an array element, the corresponding parameter in the subroutine is identified with the location of the argument so that any change to the program parameter immediately results in a change to the argument. If the argument is an expression or constant rather than a reference, the value of the argument is saved in a temporary location and this location is passed to the subroutine.

If the value of a function that takes no parameters is to be passed, the function name must be enclosed in parentheses.

```
x
a3 + rnd
y * sin(z)
(clk$)
```

## ARRAY ARGUMENTS

An array argument is written as

```
b()
```

for a vector and

```
b(,)
```

for a matrix, where b is the name of the array. The location of the array is passed to the subroutine along with the current and original array bounds. Any change to an element of the parameter array from within the subroutine immediately results in a change to the corresponding element in the argument array. The subroutine can change the current bounds of the array.

Examples:

```
a(,)
b$()
```

## FUNCTION ARGUMENTS

A function argument consists of the name of a BASIC or user-defined function. A use of the function from within the called subroutine must provide the correct number and type of arguments. Any names in the body of a user-defined function that are not function parameters or local variables of the function refer to the corresponding objects in the program in which the function is defined.

Examples:

```
sin
fnz$
```

## FILE ARGUMENTS

A file argument is written as:

```
# n
```

where n is a file expression. The file parameter in the called subroutine refers to the same file as the calling program; the file type, length, margin, pointer, and contents at entry to the subroutine remain as they were after the last operation affecting the file in the calling program. Any change to the file from within the called subroutine is retained after the subroutine returns.

Examples:

```
#n
# 3
```

## Interlanguage Calls

Calls between BASIC programs and programs written in other languages are subject to restrictions on the types of arguments that can be passed; functions, files, and arrays of strings cannot be passed. See Appendix B for further details.

## Call Statement Examples

The following are examples of the call statement:

```
100 call "init"
200 call a$ & "routine": a()
300 call "write": #k, a$(,)
400 call "integrate": fna, 1, 10, 1e-5
500 call "calculate": a, b(), sin(x-y/z)
```

## CHANGE STATEMENT

Syntax:

```
change n to s$
  or
change e$ to n
```

where n is a numeric vector, s$ is a string reference, and e$ is a string expression.

Semantics:

The change statement converts a string, considered a list of characters, to and from a numeric vector, considered a list of integers. The first form of the change statement sets the string variable to a string value whose length is given by the number found in the zero element of the numeric vector; the kth character of the string value is the character whose ASCII code is given by the kth element of the numeric vector. The second form of the change statement sets the values in the numeric vector so that the zeroth element of the vector is the length of the string expression and the kth element of the vector contains the numeric equivalent for the kth character in the string (see Appendix A).

```
100 change a to b3$
125 change seg$(a$,i+1,j-2) to b
```

## Change Bit Statement

Syntax:

```
change n to s$ bit m
  or
change e$ to n bit m
```

where n is a numeric vector, s$ is a string reference, e$ is a string expression, and m is a numeric expression. m represents a bit width; it is truncated to an integer value m', which must be from 1 to 27, inclusive.

Semantics:

The change-bit statement converts a string, considered a sequence of bits separated into pseudocharacters of length m', to and from a numeric vector, considered a list of integers. The first form of the change-bit statement packs the elements of the numeric list into a string value with each element occupying m' bits; the length of the string is the number of 9-bit characters required to hold the number of pseudocharacters specified by the zeroth element of the numeric array. The second form of the change-bit statement sets the kth element of the numeric vector to the integer value corresponding to the kth pseudocharacter in the string value; the zeroth element in the numeric vector is set to the number of pseudocharacters in the string expression.

When changing from a string to a vector, the last few bits of the string are ignored if the string is not composed of an integral number of pseudocharacters. When changing from a vector to a string, enough 0 bits are appended to the string of pseudocharacters to make an integral number of ASCII characters. When m' = 9, the change-bit statement is equivalent to the change statement.

```
10 change a to c$(a+j) bit 2*n+1
20 CHANGE a$ & b$ to x BIT 4
```

## DATA STATEMENT

Syntax:

```
data list
  or
data list ,
```

where list is a list of BASIC numeric or string constants separated by commas. A trailing comma is ignored.

Semantics:

The data statement is a nonexecutable statement that creates a block of data to be read by a read statement; 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 in which the data statements occurred, are combined to create two blocks of data: a numeric block and a string block. Numeric and string constants can be freely intermixed in a data statement; each value is entered in the appropriate data pool.

In certain cases the quotation marks around a string constant appearing in a data statement can be removed. If a string constant does not contain any blanks, tab characters, uppercase characters, backslash characters, or nonprinting ASCII characters, and does not begin with a digit, a plus sign, or a minus sign, the quotation marks can be omitted. It is never incorrect to place quotation marks around string data.

```
100 data 1,2
200 data 3.1415,2.783,0,
300 DATA these,are,unquoted,strings
400 data "This is a quoted string",1,2, "George"
```

## DEF STATEMENT

### Single Line Functions

Syntax:

```
    def f = e
     or
    def f(list) = e
```

where f is the name of the function to be defined, e is an expression, and list is a list of parameters separated by commas.

Semantics:

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

A parameter is a scalar variable that is local to the function body and has no relationship to any variable of the same name used elsewhere in the program. The value of a function parameter is initialized to the the value of the corresponding function argument when the function is called. A use inside the function body of a variable that is not a parameter refers to the variable of the same name defined outside the function definition.

A def statement can appear anywhere in the program, either before or after the first use of the function. A def statement has no effect when encountered during program execution.

```
100 def fnp = 3.14159
110 def fna(x) = sqr(1 - a * x * fnp)
120 def fnb(a$,b$) = abs(len(a$) - len(b$))
130 def fna$(a$,b$) = sst$(a$,1,pos(a$,b$,1)-1)
```

### Multiple Line Function

Syntax:

```
    def f
     or
    def f v
     or
    def f(list)
     or
    def f(list)v
```

where f is the name of the function to be defined, list is a list of parameters separated by commas, and v is a list of local variables separated by commas.

Semantics:

This form of def statement is the first statement in a multiple line function definition that is terminated by a subsequent fnend statement. The body of the function can contain any number of statements; a def statement, sub statement, or subend statement cannot be used in a function body.

Within the body of the function, the name of the function can be used as a scalar variable of the same type as the function; its value is always the value to be returned by the function. The value of the function is initialized to 0 or the null string when the function is entered; a let statement (or any other statement that can cause a variable to change its value) can be used to assign a new value for the function. Some examples of this special use of the function name are:

```
100 let fnf = 2 * fnf
120 input fnx$
```

A parameter is a scalar variable that is local to the function body. The value of a function parameter is initialized to the value of the corresponding function argument when the function is called. A change in the value of a function parameter does not cause a change in the value of the corresponding function argument.

A local variable (as specified in the def statement) is initialized to 0 or the null string; the values of the local variables are lost when the function returns. Any variable used in the function that is not a parameter or a local variable refers to the variable of the same name defined outside the function definition.

A function definition can occur anywhere in the program, either before or after the first use of the function. When a def statement is encountered during program execution, execution continues with the statement following the matching fnend statement. It is not possible to jump into or out of the function body; the function can only be invoked by a function call.

```
100 def fng(x,y)
130 def fnx$(a,b$),x,y$,z
150 def fnh
```

## DIM STATEMENT

Syntax:

```
    dim list
```

where list is a list of array declarators separated by commas. Each array declarator has the form:

```
    c(b1)
```

for vectors and:

```
    c(b1,b2)
```

for matrices; c is the name of an array; b1 and b2 are unsigned integer constants that specify the upper bound of the corresponding array dimension.

Semantics:

The dim statement explicitly declares array names and establishes the number of dimensions and number of elements in each dimension of the arrays specified. The lower bound of each array dimension is always 0. The b1 + 1 locations are reserved for a vector; (b1 + 1) * (b2 + 1) locations are reserved for a matrix.

Any number of dim statements can appear anywhere in the program; a dim statement has no effect when executed. A dim statement can appear either before or after the first use of an array; if the dim statement appears afterwards, the number of dimensions given in the dim statement must agree with the number of dimensions determined from the context of the first usage. An array name can be explicitly declared only once in a program.

```
    100 dim a(12),b(100,100),c$(23),e(3)
```

## END STATEMENT

Syntax:

```
    end
```

Semantics:

The end statement indicates the end of a main program. If there are no subroutines in the same segment, the end statement must be the last statement in the source. If subroutines are present, they must follow the end statement.

Executing an end statement causes the main program to finish normally. All files opened by the main program are closed and control returns to the program that invoked the main program.

```
    999 end
```

## FILE STATEMENT

Syntax:

```
    file # n : s$
```

where n is a file expression and s$ is a string expression.

Semantics:

The file statement opens the file with name s$ and assigns it file number n. Any file previously assigned file number n is closed. File number 0, which refers to the user's terminal, cannot be used in a file statement.

```
    100 file #1: "alpha"
    200 FILE #m: a$(i+2)
```

## FNEND STATEMENT

Syntax:

```
    fnend
```

Semantics:

The fnend statement marks the end of a multiple line function definition. See the description of the def statement.

```
175 fnend
```

## FOR STATEMENT

Syntax:

```
for v = e1 to e2
  or
for v = e1 to e2 step e3
```

where v is a reference to a scalar numeric variable, and e1, e2, and e3 are numeric expressions.

Semantics:

The for statement marks the beginning of a for-next loop; it is always used in conjunction with a subsequent next statement that specifies the same scalar numeric variable. When the optional step expression e3 is omitted, the value +1 is used.

The group of statements between the for statement and the matching next statement, called the body of the loop, is executed repeatedly according to the following steps:

1. The expressions e1, e2, and e3 are evaluated and the resulting values are saved. Let e1', e2', and e3' represent the saved values, which are inaccessible to the user's program.

2. The control variable v is set to the value of expression e1'.

3. If e3' >= 0 and v > e2' or if e3' < 0 and v < e2', the loop is terminated and execution continues with the statement after the matching next statement; otherwise, execution continues with step 4.

4. The body of the for-next loop is executed.

5. When the next statement that marks the end of the for-next loop is executed, the control variable v is set to v + e3' and step 3 is repeated.

The value of the control variable can be modified by statements within the body of the loop, and its value is available at the end of the loop. The body of the loop can contain statements that jump out of the loop, but undefined results can occur if a statement outside the for-next loop attempts to jump into the body of the loop.

For-next loops can be nested to a depth of eight. For-next loops cannot be interleaved. A for-next loop cannot use the same control variable as a for-next loop that contains it.

```
    100 for i = 1 to 10
    200 for a1 = -y to y+10 step .1
    300 for x = n to -3 step -1
```

## GOSUB STATEMENT

Syntax:

    gosub ln

where ln is a line number.

Semantics:

A gosub statement saves the line number of the statement following it and transfers control to the statement whose line number is specified in the gosub statement. When a return statement is subsequently executed, control returns to the statement whose line number was saved.

There can be any number of gosub statements executed before a return statement; the BASIC runtime system maintains a last-in first-out stack of pending returns. Any pending gosub returns that originated in a program or user-defined function are discarded when control leaves the program or function.

```
    173 gosub 1000
```

## GOTO STATEMENT

Syntax:

    goto ln

where ln is a line number.

Semantics:

The goto statement causes execution to continue with the statement whose line number is ln.

```
100 goto 75
200 go to 400
```

## IF STATEMENT

Syntax:

```
if e1 rel e2 then ln
  or
if e1 rel e2 goto ln
```

where e1 and e2 are either both numeric expressions or both string expressions, ln is a line number, and rel is one of the following relational operators:

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

Semantics:

If e1 and e2 satisfy the relationship specified by rel, control is transferred to the statement with line number ln; otherwise, execution continues with the statement following the if statement.

```
100 if a$ = "yes" then 125
200 IF ABS(X-Y) < E THEN 75
307 if x <> 0 goto 999
500 if a$ <= b$ then 200
```

## IF-END STATEMENT

Syntax:

```
    if end # n then ln
      or
    if end # n goto ln
```

where n is a file expression and ln is a line number.

Semantics:

Control is transferred to the statement with line number ln if there are no more data items to be input from the terminal format file with file number n; otherwise, execution continues with the statement following the if statement.

```
125 if end # 1 then 200
```

## IF-MORE STATEMENT

Syntax:

```
    if more # n then ln
      or
    if more # n goto ln
```

where n is a file expression and ln is a line number.

Semantics:

Control is transferred to the statement with line number ln if there are more data items available for input from the terminal format file with file number n; otherwise, execution continues with the statement following the if statement.

```
2175 if more #n goto 200
```

## INPUT STATEMENT

Syntax:

```
    input list
      or
    input list ,
```

where list is a list of references separated by commas.

Semantics:

The input statement causes data values to be read from the user's terminal and assigned, in order, to the references in the input list. The subscript expressions in an array reference in the input list are not evaluated until all references that precede it in the input list have been assigned values.

When the input statement is executed, the BASIC runtime system prints the prompt "?  " on the user's terminal without a newline as an indication that input is required. The user must enter a set of numeric or string constants separated by commas. If too few data values are provided, the BASIC runtime system prints a message and requests more data. If too many data items are provided, the BASIC runtime system prints a message and ignores the excess values.

The data items provided by the user must match the type of the corresponding reference in the input list. If the data types do not match or there is any other error, a message is printed and the incorrect data value and all values following it must be retyped.

When a numeric value is expected, the BASIC runtime system gathers all characters up to the next comma or newline; blanks and tabs are ignored. If the resulting string of characters is the word stop, execution of the program containing the input statement is terminated; otherwise, the string must be a legal BASIC numeric constant.

Either a quoted or an unquoted string can be used when a string value is expected. Quotes are necessary only if the string value is to begin with blanks or contain a comma or quote. Unlike unquoted string constants written explicitly in the BASIC program, uppercase characters are not converted to lowercase.

If the input statement ends with a comma, any data values that remain after the input list has been satisfied are not discarded, but are saved for the next statement that requests input from the terminal. When the next input statement is executed, no prompt is printed until all of the saved data values have been processed. The saved data values are discarded if a print statement is executed before this or some other statement that requests input from the user's terminal is executed.

```
100 input a,b,a$
200 input n,a(n)
300 input c$,x1,
325 input c(i,j)
```

## INPUT-FILE STATEMENT

Syntax:

```
input # n : list
  or
input # n : list ,
```

where n is a file expression and list is a list of references separated by commas.

Semantics:

This variation of the input statement requests input from the terminal format file with file number n. If the file number is 0, this form of the input statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, as many lines as are necessary to satisfy the input list are read from the specified file starting at the current value of the file pointer. No prompting messages are printed. Any erroneous input or an attempt to read past the end of the file causes a message to be printed after which execution of the program containing the input-file statement is terminated.

After each data value in the file has been processed, the file pointer is advanced to the character after the comma or newline that delimited the data value.

Like the input statement, if the input-file statement ends with a comma, any data values that remain after the input list has been satisfied are not discarded, but are saved for the next statement that requests input from the terminal. When the next input-file statement is executed, no prompt is printed until all of the saved data values have been processed. The saved data values are discarded if a print-file statement is executed before this or some other statement that requests input from the user's terminal is executed.

```
100 input #1: a,b,c
223 INPUT # k+2: n,a(n-1),
317 input #0:i,j$
```

## LET STATEMENT

Syntax:

```
    let v = e
      or
    let v1 = v2 = ... = vn = e
      or
    v = e
      or
    v1 = v2 =...= vn = e
```

where v, v1, v2, ..., vn are either all numeric references or all string references and e is an expression of the same type as the reference(s).

Semantics:

The let statement assigns the value of an expression to one or more scalar variables or subscripted array elements of the same type. All subscript expressions in the list of references are calculated before the expression is evaluated and before any assignments are done.

```
100 let x(5) = sqr(q + y^3)
217 let i = i + 1
345 a$ = b$ & seg$(c$,i,j)
400 i = a(i) = 5
```

## LINPUT STATEMENT

Syntax:

```
    linput list
```

where list is a list of string references separated by commas.

Semantics:

The linput statement causes each string reference in the list to be assigned a string value consisting of all the characters in a line of input (except the newline character at the end). This permits the user to enter strings containing characters that might otherwise have special significance to BASIC.

Each time a string value is required, a prompt is printed and an entire line is read and used for the string value. If the last input- or mat-input statement ended in a comma and there is a partial line left, the initial prompt is omitted and the partial line is used as the first string value.

```
    300 linput a1$, b$(i+3)
```

## LINPUT-FILE STATEMENT

Syntax:

```
    linput # n : list
```

where n is a file expression and list is a list of string references separated by commas.

Semantics:

This variation of the linput statement requests lines of input from the terminal format file with file number n. If the file number is 0, this form of the linput statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, as many lines as are necessary to satisfy the list of references are read from the specified file starting at the current value of the file pointer. No prompting messages are printed. If a previous input- or mat-input statement referencing the same file ended in a comma and there is any partial input line left, the value of the first string reference is set to the partial line. The file pointer is left pointing at the character after the newline of the last line read from the file.

```
    123 linput #12 : a4$
```

## MARGIN STATEMENT

Syntax:

```
    margin e
```

where e is a numeric expression.

Semantics:

The margin statement sets the maximum number of characters that can be printed on the user's terminal. The value of the expression is truncated to yield an integer value for the new margin. The new margin takes effect immediately, even if there is a partially constructed output line. The margin value lasts for the lifetime of the process or run unit, or until another margin statement in any BASIC program is executed. The default margin is 75.

```
    20 margin 120
```

## MARGIN-FILE STATEMENT

Syntax:

```
    margin # n : e
```

where n is a file expression and e is a number expression.

Semantics:

This variation of the margin statement sets the margin of the specified file to the truncated value of the numeric expression. In the case of a terminal format file, the new margin takes effect immediately; the margin of a random access file can be changed only if the file is empty. If the file number is 0, this form of the margin statement is the same as the simpler form in which the file number is omitted.

```
40 margin #6: m
```

## NEXT STATEMENT

Syntax:

```
next v
```

where v is a reference to a scalar numeric variable.

Semantics:

The next statement marks the end of a for-next loop; it is always used in conjunction with a preceding for statement that specifies the same scalar numeric variable.

```
710 next x
```

## ON-GOSUB STATEMENT

Syntax:

```
    on x gosub l1, l2, ..., ln
```

where x is a numeric expression and l1, l2, ..., ln are line numbers.

Semantics:

The on-gosub statement uses the value of the numeric expression to select one of the line numbers as the target of a gosub operation. The value of the expression is truncated to yield an integer x' that must be greater than 0 and less than or equal to the number of line numbers. The line number of the statement following the on-gosub statement is saved on the gosub return stack. If x' is 1, execution continues with line l1; if x' is 2, execution continues with line l2; and so forth.

```
220 on i gosub 1000,2000,3000,4000
```

## ON-GOTO STATEMENT

Syntax:

```
    on x goto l1, l2, ..., ln
      or
    on x then l1, l2, ..., ln
```

where x is a numeric expression and l1, l2, ..., ln are line numbers.

Semantics:

The on-goto statement uses the value of the numeric expression to select one of the line numbers as the target of a goto operation. The value of the expression is truncated to yield an integer x' that must be greater than 0 and less than or equal to the number of line numbers specified. If x' is 1, execution continues with line l1; if x' is 2, execution continues with line l2; and so forth.

```
100 on sgn(x-y)+2 goto 110,120,130
```

## PRINT STATEMENT

Syntax:

```
print
 or
print list
```

where list is a list of optional print elements separated by commas or semicolons. A print element can be an expression, a tab request, or a space request. Any number of consecutive commas or semicolons can be present in the list.

Semantics:

The print statement generates lines of output to be printed on the user's terminal. A single print statement can generate one line, several lines, or only part of a line of output. The characters generated by a print statement are sent to the terminal at the end of the statement, even if this means that the terminal print head is left sitting in the middle of a line.

The format of the line image is determined by the elements in the print list. Each element in the print list is evaluated to yield a string of characters to be placed in the output line. If the resulting string fits on the current line, the characters are appended to the partial line.

If the string of characters corresponding to the print element would cause the length of the current output line to exceed the margin, a newline is placed at the end of the line; the remainder of the line is transmitted to the terminal; and a new line is begun. If the string of characters derived from the print element is longer than the margin, the string is split across as many complete lines as are required to hold it.

### Numeric Expressions

A numeric expression appearing in a print list is evaluated and converted to a character string representation. One of three formats is used for the string representation, depending on the characteristics of the numeric value. In all cases, the string begins with a sign character and ends with a blank; the sign is a blank if the value is positive and a minus sign if the value is negative.

## INTEGER FORMAT

Numbers printed in integer format consist of a string of from one to n decimal digits without a decimal point where n is nine for single precision and nineteen for double precision. Integer format is used for integers whose absolute value is less than 2^27 (134,217,728) for single precision and less than 2^63 (9223372036854775808) for double precision. Some examples of numbers in integer format are:

```
12
-20765
0
```

## FRACTIONAL FORMAT

Numbers printed in fractional format consist of from one to six decimal digits with a decimal point. Trailing 0's in the fractional part are omitted; a number less than 1 is represented with a zero to the left of the decimal point. Numbers printed in fractional format are rounded to six digits. Fractional format is used for nonintegers whose absolute values are in the range 0.0999995 to 999999.5 or whose absolute values are less than 0.0999995 and can be exactly represented with six digits. In double precision mode, the number of digits may be specified by the setdigits statement. Some examples of numbers in fractional format are:

```
12.34
-0.00276
0.0037
7.13486
```

## SCIENTIFIC FORMAT

A number printed in scientific format appears as:

x E+y   or   x E-y

where x is a number whose absolute value is greater than or equal to 1 and less than 10 printed in fractional notation and y is a power of 10 such that the numeric value being converted is x * 10 ^ y. Numbers printed in scientific format are rounded to six digits. In double precision mode, the number of digits may be specified by the setdigits statement. Scientific format is used whenever integer or fractional format cannot be used. Some examples of numbers printed in scientific format are:

```
-7.31567 E+13
1.27 E-21
-1. E-32
```

## String Expressions

A string expression in the print list is evaluated and the resulting string of characters is placed in the output line. The BASIC runtime system does not look at the contents of the character string; unpredictable results can occur if the string contains characters that do not advance the print head by precisely one position.

## Comma Separator

The output line is normally considered to be divided into zones of 15 characters each. The first zone starts in column 0, the second zone starts in column 15, and so forth. The zone after the last zone on a line is the first zone on the next line. The number of zones available is determined by the current value of the margin; the default margin value of 75 permits five zones.

A comma in a print list causes the print head to advance to the beginning of the next available zone; this can cause the current line to be printed and a new line to be started. If a comma is the last element in the print list, the partial line, if any, is printed and the print head remains positioned at the start of the new zone.

## Semicolon Separator

A semicolon in a print list is used only to separate print elements and does not affect the position of the print head. This permits the elements on either side of the semicolon to be printed without any extra spaces between them. If a semicolon is the last element in the print list, the partial line, if any, is printed and the print head remains positioned at the character after the last character printed.

## Tab Request

The tab print element requests that the print head be moved to a specific column. A tab request is written as:

```
tab(e)
```

where e is a numeric expression. The truncated value of the expression is taken modulo the current margin value to yield a value e' for the desired column. If the print head is already past column e', nothing happens; otherwise, the print head is positioned at column e'. Since a comma in the print list advances the print head, a tab request should normally be followed by a semicolon.

An example of a tab request in a print list is:

```
x; tab(40); y
```

## Space Request

The space print element requests that the print head be advanced by a specific number of columns. A space request is written as:

```
spc(e)
```

where e is a numeric expression. The value of the expression is truncated to yield an integer e' that gives the number of spaces desired. If the specified number of spaces would take the print head past the margin, the space request is ignored; otherwise, the print head is advanced.

## List Termination

If the print list does not end in a comma or a semicolon, a newline character is appended to the line and the line is transmitted to the terminal. A completely empty print list causes the previous line to be finished or a blank line to be printed.

## Print Statement Examples

The following examples show the optional print elements and separators described in the previous paragraphs:

```
10 print
20 print x,sin(z^2 - y^2)
30 print "Value is ";x-y
40 print ,,,a$ & seg$(b$,i,j)
50 print x;
60 print x,y,
```

## PRINT-FILE STATEMENT

Syntax:

```
    print # n
      or
    print # n : list
```

where n is a file expression and list is a list of optional print elements separated by commas or semicolons. A print element can be an expression, a tab request, or a space request. Any number of consecutive commas or semicolons can be present in the list.

Semantics:

This variation of the print statement directs output to the terminal format file with file number n. If the file number is 0, this form of the print statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, lines are written into the file starting at the current position of the file pointer. Each time a partial line is written into the file, the file pointer is updated to point to the character after the last character written.

```
100 print #3: x,fna(x)
```

## PRINT-USING STATEMENT

Syntax:

```
    print using f$
      or
    print using f$ ;
      or
    print using f$, list
      or
    print using f$, list ;
```

where f$ is a string expression and list is a list of expressions separated by commas.

Semantics:

The print-using statement generates lines of output to be printed on the user's terminal. A single print-using statement can generate one line, several lines, or only part of a line of output. The characters generated by a print-using statement are sent to the terminal at the end of the statement, even if this means that the terminal print head is left sitting in the middle of a line.

## Format Fields

The string specified by f$ contains a description of the editing to be applied to the values in the print list. The format string f$ is divided into a series of fields, each of which controls the formatting of a single value in the print list. Two types of fields are possible: numeric fields and string fields. A numeric field can only be used with a numeric value and a string field can only be used with a string value.

There are eight special characters used for defining fields in the format string. These characters and their effects are given in the following table:

| Character | Effect |
|---|---|
| - | Start a numeric field; print a floating minus sign for negative numbers and reserve a place for a digit for positive numbers. |
| + | Start a numeric field, print a floating plus sign for positive numbers and a floating point minus sign for negative numbers. |
| . | Mark the position where a decimal point is to be printed. |
| $ | Start a numeric field; print a floating dollar sign. |
| - | Specify the exponent part of a numeric field. |
| < | Start a string field; print string left justified. |
| > | Start a string field; print string right justified. |
| # | Reserve a place in either a numeric of a string field. |

A format field consists of all of the characters from the character that starts the field until the end of the format string or the character before the character that starts the next field, whichever comes first.

A character that is not one of the eight special format characters is called a literal character. Literal characters occurring in a field are normally placed in the line image unchanged; they can be replaced by blanks as described below under "Numeric Fields".

1. A "+" or a "-" can be immediately preceded by a "$".

2. If a "$" is not immediately followed by a "+" or "-", "-" is assumed.

3. The exponent field must be written as "^^^^^".

4. A "#" cannot start a field.

5. A "." is a literal character when it occurs outside of a numeric field.

The following are examples of format strings:

```
"x is -## and f(x) is +##.##^^^^^"
"RECEIPTS $-##,###.00"
"<#####               >########"
```

In the first example, the string "x is " precedes the first field which consists of the string "-## and f(x) is "; the second field consists of the string "+##.##^^^^^".

## Format Processing

The print-using statement is processed in the following manner:

1. The optional string of literal characters that precedes the first field in the format string is placed in the line image with normal margin checking.

2. Each expression in the print list is evaluated, in turn, and its value is used to evaluate the corresponding field in the format string. The string of characters resulting from the evaluation of the format field is placed in the line image.

3. If there are more format fields than expressions in the print list, the extra fields are ignored and processing ceases.

4. If the end of the format string is reached before the last expression has been evaluated, a newline is added to the current output line, the line is transmitted to the

terminal, and the entire format string is used again, starting with the first field.

5. If the print list does not end in a semicolon, a newline is added to the line image. The line is transmitted to the terminal.

6. If the string of characters resulting from a field evaluation would cause the margin to be exceeded, characters are placed in the line until the margin is exactly reached, a newline is appended, the characters are sent to the terminal, and the remaining characters are placed in the following line. As many lines as are necessary to hold the field value are used.

## Numeric Fields

A numeric field that does not contain an exponent part is evaluated as follows:

1. Each "#" in the field reserves a place for a digit in the converted numeric value. A "-" reserves a place for a digit if the numeric value is positive. Let P be the number of digit places reserved.

2. If the field does not contain a ".", the numeric value is converted with rounding to a decimal integer of P digits. If the field contains a "." followed by Q digit places, the numeric value is converted to a decimal number of P digits and is rounded to Q fractional digits. Let D be the string of digits obtained.

3. If P = 0, or Q > 38, or any high-order digits are lost in the conversion performed by step 2, the value of the field is the string of characters in the field with each "+", "-", "$", and "#" replaced by a "*".

4. Leading 0's in the integer part of D are replaced by blanks, except a blank is never placed in the units position. If more than n digits remain after leading 0's have been removed, all digits after the nth digit are replaced by a "?". n is nine for single precision but may be specified by the setdigits statement for double precision.

5. A copy is made of the characters in the field with each character in the field that reserved a digit position being replaced by the corresponding character of D. Let C be the string thus obtained.

6. All characters in C to the left of the first digit that is not a 0 are replaced by blanks.

7. If the field contains a "+", the sign of the numeric value replaces the blank immediately preceding the leftmost digit of C. If the field contains a "-" and the numberic value is positive, no action is taken. If the field contains a "-" and the numeric value is negative, a "-" replaces the blank just preceding the leftmost digit, or, if there is no blank, a leading 0 in the units position.

8. If the field begins with a "$", a "$" replaces the blank just preceding the sign or leftmost digit of C.

9. The value of the field is the string obtained in step 8.

A numeric field that contains an exponent part is evaluated as follows:

1. Each "#" in the field reserves a digit position. Let P be the number of digit positions.

2. If the field does not contain a ".", let Q be 0; otherwise, let Q be the number of digit positions to the right of the ".". The numeric value is evaluated and converted with rounding to a decimal value D of P digits with Q fractional digits and exponent E such that the original value is D * 10 ^ E. If the value of the expression is not 0, the leftmost digit of D is not 0.

3. A copy is made of the characters in the field with each character that reserved a digit position being replaced by the corresponding digit of D. Let C be the copy obtained.

4. A "+" in C is replaced by the sign of the numeric value; a "-" in C is replaced by a space if the numeric value is positive.

5. The first "^" in the exponent field is replaced by a blank; the second "^" is replaced by an "E"; the third "^" is replaced by the sign of the exponent E; and the last two "^"s are replaced by the decimal exponent digits. If the exponent can be represented by a single digit, the last character of the exponent field is replaced by a blank.

6. The value of the field is the string obtained in Step 5.

Some examples of numeric field evaluation are presented here (ƀ represents a single blank):

| Field | Internal Value | External Form |
|---|---|---|
| -## | 2 | ƀƀ2 |
| -## | -23 | -23 |
| -## | 476 | 476 |
| -## | -476 | *** |
| +## | 23 | +23 |
| +## | -23 | -23 |
| +## | 476 | *** |
| +## | 0 | ƀ+0 |
| -##.## | 17.479 | ƀ17.48 |
| -##.## | 1.7479 | ƀƀ1.7 |
| -##.## | .17479 | ƀƀ0.17 |
| -##.## | -.172 | ƀ-0.17 |
| -.## | 0.23 | 0.23 |
| -.## | -0.23 | -.23 |
| - | 7 | 7 |
| - | -7 | * |
| $-#,###.00 | 18.43 | ƀƀƀƀ$18.00 |
| $-#,###.00 | -1234 | $-1,234.00 |
| -##.##^^^^^ | 123.4 | ƀ12.34 E+1ƀ |
| -##.##^^^^^ | -1.234e14 | -12.34 E+13 |
| -##.##^^^^^ | 0 | ƀ00.00 E+0ƀ |

## String Fields

A string field is evaluated as follows:

1. Each "#" in the field reserves a character position as does the "<" or ">" with which the field begins. Let P be the number of places reserved.

2. The character string expression is evaluated. Let S be the string resulting from the evaluation, and let N be the number of characters in S.

3. If the field starts with "<", the field is copied from left to right. The "<" is replaced by the leftmost character of S; each "#" is replaced by the next character of S in sequence from left to right. If N > P, the excess N - P characters are dropped from the right end of S. If N < P, the last P - N character positions in the field are replaced by blanks. Any literal character in the field is copied without change.

4. If the field starts with ">", the field is copied from right to left. The rightmost "#" in the field is replaced by the rightmost character of S; each "#" and the ">" are replaced by the next character of S in sequence from right to left. If N > P, the excess N - P characters are dropped from the left end of S. If N < P, the first P - N character positions are replaced by blanks. Literal characters in the field are copied without change.

5. The value of the field is the string resulting from Step 3 or Step 4.

The following are some examples of string field evaluation (ƀ indicates a single blank):

| Field | Internal Value | External Form |
|---|---|---|
| <######## | alpha | alphaƀƀƀƀ |
| >######## | beta | ƀƀƀƀƀbeta |
| >######## | betaƀ | ƀƀƀƀbetaƀ |
| <## | alpha | alp |
| >## | alpha | pha |
| <1#2#3#4# | alpha | a1l2p3h4a |

## Printing Special Characters

If the user wishes to print a literal copy of one of the eight characters with special meaning in format fields, he must use a string field and pass the character as part of the print list. For example, the following statement prints a period at the end of the sentence:

```
    100 print using "x is -###<", x, "."
```

If the statement had been written:

```
    100 print using "x is -###.", x
```

the "." would be treated as part of the numeric field.

## Print-Using Statement Examples

The following examples illustrate the use of the print-using statement:

```
100 print using f$, a, b, c
200 print using "-##.##", sqr(x);
300 print using "<hits = -###", "#", h
```

## PRINT-FILE-USING STATEMENT

Syntax:

```
print #n: using f$
  or
print #n: using f$ ;
  or
print #n: using f$, list
  or
print #n: using f$, list ;
```

where n is a file expression, f$ is a string expression, and list is a list of expressions separated by commas.

Semantics:

This variation of the print-using statement directs output to the terminal format file with file number n. If the file number is 0, this form of the print-using statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, lines are written into the file starting at the current position of the file pointer. Each time a partial line is written into the file, the file pointer is updated to point to the character after the last character written.

```
100 print #3: using "x is <###", a$
```

## RANDOMIZE STATEMENT

Syntax:

```
    randomize
```

Semantics:

The randomize statement initializes the pseudorandom number generator (of the program containing the randomize statement) with a value derived from the Multics clock reading at the time the randomize statement is executed. If a program does not contain a randomize statement, the same sequence of pseudorandom numbers is generated each time the program is executed. The randomize statement affects only the program in which it occurs.

```
100 randomize
```

## READ STATEMENT

Syntax:

```
    read list
```

where list is a list of references separated by commas.

Semantics:

The read statement causes values from the data pools, starting at the next available values, to be assigned, in order, to the references in the list. Numeric references are assigned values from the numeric data pool, and string references are assigned values from the string data pool. The subscript expressions in an array reference in the read list are not evaluated until all references that precede it in the read list have been assigned values.

```
100 read x,f(x)
120 read a$, b$, x(3)
```

## READ-FILE STATEMENT

Syntax:

```
    read # n : list
```

where n is a file expression and list is either a list of numeric references separated by commas or a list of string references separated by commas.

Semantics:

This variation of the read statement reads from the random file with file number n. The type of file n must be the same as the type of the references in the list. The file number cannot be 0.

The read-file statement reads values from the random file starting with the data item pointed at by the file pointer. The file pointer is incremented by 1 after each data value is read.

```
100 read #3: v$(i),a$
110 read #4: a,b,c
```

## REM STATEMENT

Syntax:

```
    rem S
```

where S is any sequence of ASCII characters that does not include the newline character.

Semantics:

The string of characters following "rem" is ignored by the BASIC compiler. The rem statement has no effect when encountered during execution.

## RESET STATEMENT

Syntax:

```
    reset
```

Semantics:

The reset statement reinitializes the data pools so that the next read statement executed reads the first data item in the appropriate pool. The reset statement resets both the numeric pool and the string pool. Only the program that contains the reset statement is affected.

```
900 reset
```

## RESET-FILE STATEMENT

Syntax:

```
reset #n
  or
reset #n: m
```

where n is a file expression and m is a numeric expression.

Semantics:

This variation of the reset statement resets the file pointer of the file with file number n. The file number cannot be 0.

If the expression m is omitted, file number n must be a terminal format file; in this case, the file pointer is reset to the beginning of the file. If the expression m is present, file number n must be a random file; in this case, the file pointer is set to point at the data value whose position in the file is given by the integer part of m.

```
100 reset #1
120 reset #k: j-2
```

## RETURN STATEMENT

Syntax:

```
return
```

Semantics:

The return statement causes control to return to the statement following the most recently executed gosub statement executed by the program that contains the return statement.

```
199 return
```

## SCRATCH STATEMENT

Syntax:

    scratch #n

where n is a file expression.

The scratch statement erases the current contents, if any, of the file with file number n and positions the file pointer at the beginning of the file. The file can be of any kind; the file number must not be 0.

```
100 scratch #k
```

## SETDIGITS STATEMENT

Syntax:

    setdigits n

where n is a numeric expression.

The setdigits statement specifies the number of digits to be printed by all future print statements until another setdigits statement is executed, or until the end of program execution. The value of n can be 1 through 19. This statement applies only to nonintegers. The tab spacing is adjusted to accomodate the current number length. However, the spacing is never less than the default.

The setdigits statement takes effect only in double precision mode.

```
110 setdigits k
```

## STOP STATEMENT

Syntax:

```
    stop
```

Semantics:

The stop statement causes the program in which it occurs to return to its caller; any files opened by the program are closed. Executing a stop statement in a main program is equivalent to executing the end statement. Executing a stop statement in a subroutine program is equivalent to executing the subend statement.

```
    999 stop
```

## SUB STATEMENT

Syntax:

```
    sub s$
      or
    sub s$ : list
```

where s$ is a string constant and list is a list of parameter specifications. A parameter specification can be a scalar parameter, an array parameter, a function parameter, or a file parameter.

Semantics:

The sub statement is the first statement in a subroutine program. The constant s$ gives the name of the subroutine being defined; it must consist of 32 or fewer characters.

A subroutine can contain any BASIC statement except the end statement. Subroutines cannot be nested; each sub statement must be paired with a subsequent subend statement. Any name or file number used in a subroutine that is not a parameter of the subroutine is local to the subroutine and has no connection with the same name or file number used in some other program or in a previous use of the same subroutine. Each program has its own pair of data pools and its own pseudorandom number generator, all of which are reset when the program is entered.

The line numbers used in a program have no relationship to line numbers used in other programs. The line number of a sub statement can be less than the line number of the preceding end statement or subend statement.

A subroutine can be entered only by means of a call statement that references the sub statement; a subroutine can be left only by means of a stop statement or the subend statement that terminates the subroutine body. It is not possible to jump into or out of a subroutine. If certain errors are detected during the execution of a subroutine, execution of the subroutine is terminated and control returns to the caller of the subroutine.

## Parameters

When a subroutine is called, the items in the optional parameter list are associated with the items in the argument list in the call statement. Any reference to a parameter results in a reference to the corresponding argument. The number and type of the arguments provided must match the number and type of the parameters expected.

Four types of parameters can be received: scalars, arrays, functions, and files.

### SCALAR PARAMETERS

A scalar parameter is a reference to a scalar variable. The value of the parameter at entry to the subroutine is the value of the corresponding argument. Any change to the parameter immediately causes a change to the corresponding argument.

### ARRAY PARAMETERS

An array parameter is written as

```
b()
```

for a vector and

```
b(,)
```

for a matrix, where b is the name of the array. The number of dimensions of the parameter array must be the same as the number of dimensions of the argument array. The parameter array cannot be dimensioned in the subroutine with a dim statement; the parameter array uses the original and current bounds of the argument array. Any redimensioning of the parameter array immediately affects the argument array; any change to an element of the parameter array immediately affects the corresponding element of the argument array. (See Section 6 for the definition of redimensioning.)

## FUNCTION PARAMETERS

A function parameter consists of the name of a user-defined function. The type (numeric or string) implied by the name of the function parameter must agree with the type of the function argument. The number and type of arguments provided in a use of the function from inside the subroutine must agree with the number and type of parameters expected by the function. The function name cannot appear in a def statement inside the subroutine. If the function that was passed to the subroutine contains references to names that are not parameters or local variables of the function, these names refer to names in the program in which the function is defined.

## FILE PARAMETERS

A file parameter is written as

#c

where c is an integer constant. File number c in the subroutine is associated with the file passed as the argument in such a way that any change to the file from within the subroutine is preserved when control leaves the subroutine. All the characteristics of the file parameter are identical to the characteristics of the file argument.

### Sub Statement Examples

The following examples of the sub statement match the examples of the call statement given earlier:

```
100 sub "init"
110 sub "x_routine": b()
120 sub "write": #2, c$(,)
130 sub "integrate": fnz, l, u, e
140 sub "calculate": a,b(),c
```

## SUBEND STATEMENT

Syntax:

```
subend
```

Semantics:

The subend statement is the last statement in a subroutine program. When the subend statement is executed, the storage allocated for variables of the subroutine is released, all files opened by the subroutine are closed, and control returns to the caller of the subroutine.

```
9000 subend
```

## TIME STATEMENT

Syntax:

```
time c
```

where c is a positive numeric constant.

Semantics:

The time statement establishes a time limit of c seconds of Multics virtual processor time for the execution of the program in which it occurs; execution of the program is terminated if the limit is exceeded. Any number of time statements can appear anywhere in the program; the limit used is the smallest limit specified. A time statement has no effect when executed.

```
10 time 3
```

## WRITE STATEMENT

Syntax:

```
write #n : list
```

where n is a file expression and list is either a list of numeric expressions separated by commas or a list of string expressions separated by commas.

Semantics:

The write statement writes into the random file with file number n. The type of the file must be the same as the type of references in the list. The file number cannot be 0.

The write statement writes the values of the expressions in the list, in order, into the random file starting with the current position of the file pointer. The file pointer is incremented by 1 after each value is written.

```
200 write #3: x, a*b - c
220 write #4: a$
```

# SECTION 6

# ARRAY STATEMENTS

The BASIC statements discussed in Section 5 permit the elements of an array to be manipulated on an element by element basis. The mat statement described in this section allows arrays to be manipulated as single entities.

All of the array statements deal with both vectors and matrices. When a matrix is used in an array input/output statement, elements are transmitted row by row. Normally the zeroth element of a vector and the zeroth row and zeroth column of a matrix are ignored by the array statements; however, in some cases (noted below) the values of these elements can be destroyed.

## ARRAY REDIMENSIONING

The original and current bounds of an array are determined by a dim statement or by the default bounds values. The current bounds can be changed by the process known as redimensioning.

Whenever an array receives as its value the contents of another array with different current bounds, the current bounds of the target array are automatically changed to be the same as the current bounds of the source array. When the current bounds of an array change, any elements of the array with one or more subscripts equal to 0 are destroyed.

The total amount of storage implied by the current bounds must be less than or equal to the total amount of storage reserved by the original bounds. For example, a matrix originally dimensioned 10 x 10 could be redimensioned 5 x 5, 4 x 20, or 30 x 2, but not 5 x 25.

In some forms of the mat statement the new value for the current bounds of an array can be explicitly stated by a bounds list of the form

```
    (b1)
```

for a vector and

```
    (b1,b2)
```

for a matrix. Both b1 and b2 are numeric expressions whose values are truncated to yield integer values for the new bounds. The bounds list is written immediately after the name of an array or array constant.

The following are all examples of valid bounds lists:

```
(n)
(n,n)
(m,n)
(m+2,m-2)
```

## ARRAY INITIALIZATION

Syntax:

```
    mat a = con
      or
    mat b = idn
      or
    mat a = zer
      or
    mat a$ = nul$
```

where a is a numeric array, b is a square numeric matrix, and a$ is a string array.

Semantics:

These array initialization statements set the array appearing to the left of the equal sign to a constant array having the same bounds. The names appearing to the right of the equal sign are called array constants.

The constant con is the array with all elements having the value 1. The constant idn is the square identity matrix I defined by

```
    I(i,j) = 1 if i = j

    I(i,j) = 0 if i <> j
```

The constant zer is the array with all elements having the value 0. The constant nul$ is the string array in which all elements are the zero-length string.

Note that while the other array constants can be used with either vectors or matrices, the array constant idn can only be used with a matrix having an equal number of rows and columns.

```
200 mat v = con
300 mat z$ = nul$
```

## ARRAY INITIALIZATION WITH REDIMENSIONING

Syntax:

```
    mat a = con bounds
      or
    mat b = idn bounds
      or
    mat a = zer bounds
      or
    mat a$ = nul$ bounds
```

where a is a numeric array, b is a square numeric matrix, a$ is a string array, and bounds is a bounds list.

Semantics:

These array initialization statements set the array appearing to the left of the equal sign to a constant array having the bounds specified by the bounds list. When the idn constant is used in this manner, the two bounds expressions must be equal.

```
100 mat a = con(4)
120 mat a$ = nul$(n,m)
140 mat b = idn(p,p)
160 mat c = zer(m-1,n+1)
180 mat d = con(5,5)
```

## ARRAY ASSIGNMENT

Syntax:

```
    mat a = b
```

where a and b are either both vectors of the same type or both matrices of the same type.

Semantics:

The array assignment statement sets the array appearing to the left of the equal sign to the value of the array appearing to the right of the equal sign. Both arrays must be of the same type and must have the same number of dimensions. The current bounds of the target array are changed to the current bounds of the source array.

```
100 mat x = y
120 mat x$ = y$
```

## ARRAY ADDITION

Syntax:

```
    mat a = b + c
```

where a, b, and c are either all numeric vectors or all numeric matrices.

Semantics:

The array addition statement sets the elements of the array appearing to the left of the equal sign to the sum of the corresponding elements of the two arrays appearing to the right of the equal sign. All of the arrays must have the same number of dimensions; the two input arrays must have the same current bounds. The current bounds of the target array are changed to the current bounds of the source arrays.

```
100 mat x = y + y
```

## ARRAY SUBTRACTION

Syntax:

```
mat a = b - c
```

where a, b, and c are either all numeric vectors or all numeric matrices.

Semantics:

The array subtraction statement sets the elements of the array appearing to the left of the equal sign to the difference of the corresponding elements of the two arrays appearing to the right of the equal sign. All of the arrays must have the same number of dimensions; the two input arrays must have the same current bounds. The current bounds of the target array are changed to the current bounds of the source arrays.

```
140 mat x = y - z
```

## ARRAY MULTIPLICATION

There are three kinds of array multiplication as follows:

### Scalar Multiplication

Syntax:

```
mat a = ( e ) * b
```

where a and b are either both numeric vectors or both numeric matrices and e is a numeric expression.

Semantics:

This form of the array multiplication statement sets the elements of the array appearing to the left of the equal sign to the value of the corresponding elements of the array appearing to the right of the equal sign with each element multiplied by the value of the numeric expression. The two arrays must have the same number of dimensions; the current bounds of the target array are changed to the current bounds of the source array.

```
300 mat a = (5) * b
320 mat b = (sqr(1-x/y))*b
```

<u>Inner Product</u>

Syntax:

```
mat v = a * b
```

where v is a numeric scalar reference and a and b are numeric vectors.

Semantics:

This form of the array multiplication statement assigns the inner or dot product of two vectors to a scalar reference. The two vectors being multiplied must have the same number of elements.

```
100 mat x(i+j) = a * b
100 mat a = b * c
```

<u>Outer Product</u>

Syntax:

```
mat a = b * x
  or
mat a = x * b
  or
mat x = y * z
```

where a and b are numeric vectors and x, y, and z are numeric matrices.

Semantics:

This form of the array multiplication statement sets the array appearing to the left of the equal sign to the outer product of the arrays appearing to the right of the equal sign. The arrays must satisfy the normal rules for matrix multiplication: when two arrays are multiplied, the number of columns in the first array must equal the number of rows in the second array; the result is an array with the same number of rows as the first array and the same number of columns as the second array.

When a vector containing N elements is multiplied by a matrix dimensioned N x P, the vector is treated as a row vector and the result is a vector with P elements. When a matrix dimensioned M x N is multiplied by a vector containing N elements, the vector is treated as a column vector and the result is a vector with M elements. When a matrix dimensioned M x N is multiplied by a matrix dimensioned N x P, the result is a matrix dimensioned M x P.

The number of dimensions of the target array must be the same as the number of dimensions of the array resulting from the multiplication. The current bounds of the target array are changed to those of the result array.

```
100 mat a = b * c
700 mat a = a * b
820 mat c = c * c
```

## TRANSPOSE FUNCTION

Syntax:

```
mat a = trn(b)
```

where a and b are either both numeric vectors or both numeric matrices.

Semantics:

The array transpose statement sets the array appearing to the left of the equal sign to the transpose of the array appearing to the right of the equal sign. The rows (columns) of the target array are the columns (rows) of the source array. The target array must have the same number of dimensions as the source array. The current bounds of the target array are changed; if the source array is dimensioned M x N, the current bounds of the target array are changed to N x M. The transpose of a vector is the vector itself.

```
900 mat a = trn(b)
920 mat c = trn(c)
```

## INVERSE FUNCTION

Syntax:

```
mat a = inv(b)
```

where a is a numeric matrix and b is a square numeric matrix.

Semantics:

The matrix inverse statement sets the matrix appearing to the left of the equal sign to the inverse of the square matrix appearing to the right of the equal sign. The current bounds of the target matrix are changed to the current bounds of the source matrix.

The numeric function det (see Section 3) returns the determinant of the matrix inverted by the last matrix inverse statement to be executed in the same program. The user must determine for himself if the determinant is large enough for the inverse to be meaningful.

If the matrix being inverted is singular, the determinant is set to 0 and execution continues without any error message being printed. In this case, the value of the target matrix is undefined.

```
100 mat a = inv(b)
200 mat c = inv(c)
```

## MAT INPUT STATEMENT

Syntax:

```
    mat input list
      or
    mat input list ,
```

where list is a list of array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

The mat-input statement causes data values to be read from the user's terminal and assigned, in order, to the elements of the arrays in the input list. The type of data provided must match the type of array being filled. The format of permissible data values, the error conditions, and the prompts used are the same as in the input statement.

In general, enough data values are read from the terminal to completely fill an array according to the current bounds of the array. If an array name in the input list is followed by a bounds list, the array is redimensioned with the specified bounds before any input values are read. If more input values are provided than are required to fill an array, the excess values are stored in the next array in the list or are ignored if there are no more arrays in the input list. If the input list ends with a comma, any data values that remain after all arrays have been filled are saved for the next input request from the terminal.

The data items provided by the user must match the type of the corresponding array element. If the data types do not match or there is any other error, a message is printed and the incorrect data value and all values following it must be retyped.

If the last array in the input list is a vector that is not followed by a bounds list, an arbitrary number of data values, from no elements to the maximum number of elements that can be held by the vector, can be provided by the user. In this case, the vector is automatically redimensioned to the number of data values provided.

If all of the data values to be input do not fit on a single line, the user can end the input line with an ampersand (&), in which case BASIC requests another line of input. Any number of lines can be continued in this manner; the ampersand can be preceded by a comma. If an arbitrary number of values is being read as described above, input is terminated by the first line that does not end in an ampersand. If a string value is to end in an ampersand, it must be enclosed in quotes.

The numeric function num (see Section 3) gives the number of data values placed in the last array by the last mat-input statement executed.

```
100 mat input a
200 mat input b(n,m),c(3,4)
300 mat input a$
400 mat input a$(k)
```

## MAT INPUT FILE STATEMENT

Syntax:

```
mat input # n : list
  or
mat input # n : list ,
```

where n is a file expression and list is a list of array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

This variation of the mat-input statement requests input from the terminal format file with file number n. If the file number is 0, this form of the mat-input statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, as many lines as are necessary to satisfy the input list are read from the specified file starting at the current value of the file pointer. No prompting messages are printed. Any erroneous input or an attempt to read past the end of the file causes a message to be printed, after which execution of the program containing the mat-input statement is terminated.

After each data value in the file has been processed, the file pointer is advanced to the character after the comma or newline that delimited the data value.

```
100 mat input #3: a,b(n)
200 mat input #m: b$(j,k)
```

## MAT LINPUT STATEMENT

Syntax:

    mat linput list

where list is a list of string array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

The mat-linput statement causes successive lines of input from the terminal to become the values of the elements of the string arrays in the linput list. Each element gets set to a string value consisting of all the characters in a line of input except for the newline character that terminates the line.

Each time a string value is required, a prompt is printed and an entire line is read and used for the string value. If the last input- or mat-input statement ended in a comma and there is a partial line left, the initial prompt is omitted and the partial line is used as the first string value.

Enough lines are read to completely fill an array according to the current bounds of the array. If an array name in the linput list is followed by a bounds list, the array is redimensioned with the specified bounds before any lines are read. It is not possible to input an arbitrary number of lines.

```
175 mat linput a$,b$
```

## MAT LINPUT FILE STATEMENT

Syntax:

    mat linput # n : list

where n is a file expression and list is a list of string array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

This variation of the mat-linput statement requests lines of input from the terminal format file with file number n. If the file number is zero, this form of the linput statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, as many lines as are necessary to satisfy the linput list are read from the specified file starting at the current value of the file pointer. If a previous input- or mat-input statement referencing the same file ended in a comma and there is any partial input line left, the value of the first array element is set to the partial line. The file pointer is left pointing at the character after the newline of the last line read from the file.

```
300 mat linput #4: a$(n)
```

## MAT PRINT STATEMENT

Syntax:

```
mat print list
  or
mat print list ,
  or
mat print list ;
```

where list is a list of array names separated by commas or semicolons.

Semantics:

The mat-print statement causes all the elements of an array with subscripts that are not 0 to be printed row by row. If a vector is not followed by a comma or a semicolon (i.e., it is the last array in the print list), it is printed as a column vector with one element per line; otherwise, it is printed as a row vector.

If an array name is followed by a semicolon, elements are printed without intervening spaces; otherwise, elements are printed in the normal 15-column zones. Each row of the array begins on a new line of output and extends over as many lines as are necessary. A blank line is printed between each array.

The rules used for formatting individual array elements and for terminating partial output lines are the same as for the print statement.

```
100 mat print a,b;
110 mat print d$,
```

## MAT PRINT FILE STATEMENT

Syntax:

```
    mat print # n : list
      or
    mat print # n : list ,
      or
    mat print # n : list ;
```

where n is a file expression and list is a list of array names separated by commas or semicolons.

Semantics:

This variation of the mat-print statement directs output to the terminal format file with file number n. If the file number is 0, this form of the mat-print statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, lines are written into the file starting at the current position of the file pointer. Each time a line is written into the file, the file pointer is updated to point to the character after the last character written.

```
100 mat print #6: a,b;
```

## MAT PRINT USING STATEMENT

Syntax:

```
    mat print using f$, list
```

where f$ is a string expression and list is a list of array names separated by commas.

Semantics:

The mat-print-using statement causes all the elements of an array with subscripts that are not 0 to be printed row by row according to the specified format string. The allowable format strings and the rules for their interpretation are the same as for the print-using statement.

The format string is scanned from the beginning for each row of the array. Each row of the array starts a new line of output. If the format string is exhausted before a row is complete, it is rescanned from the beginning. All vectors are printed as row vectors.

```
100 mat print using "-#  -#  -#", a
200 mat print using f$, a$,b
```

MAT PRINT USING FILE STATEMENT

Syntax:

```
mat print # n : using f$, list
```

where n is a file expression, f$ is a string expression, and list is a list of array names separated by commas.

Semantics:

This variation of the mat-print-using statement directs output to the terminal format file with file number n. If the file number is zero, this form of the mat-print-using statement is the same as the simpler form in which the file number is omitted.

If the file number is nonzero, lines are written into the file starting at the current position of the file pointer. Each time a line is written into the file, the file pointer is updated to point to the character after the last character written.

```
100 mat print #2: using g$, a,b
```

MAT READ STATEMENT

Syntax:

```
mat read list
```

where list is a list of array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

The mat-read statement causes values from the data pools, starting at the next available values, to be assigned, in order, to the elements of the arrays in the read list. Numeric arrays are assigned values from the numeric data pool and string arrays are assigned values from the string data pool.

Enough data values are read from the appropriate data pool to completely fill an array according to the current bounds of the array. If an array name in the read list is followed by a bounds list, the array is redimensioned with the specified bounds before any data values are read.

```
100 mat read a$,b
200 mat read c(n,m)
```

## MAT READ FILE STATEMENT

Syntax:

```
    mat read # n : list
```

where n is a file expression and list is either a list of numeric array names separated by commas or a list of string array names separated by commas. Each array name can be optionally followed by a bounds list.

Semantics:

This variation of the mat-read statement reads from the random file with file number n. The type of file n must be the same as the type of the arrays in the list. The file number cannot be 0.

The mat-read-file statement reads values from the random file starting with the data item pointed at by the file pointer. The file pointer is incremented by 1 after each data value is read.

```
100 mat read #3: a$,b$(n)
200 mat read #m: a,b,c(k,k)
```

## MAT WRITE FILE STATEMENT

Syntax:

```
mat write # n : list
```

where n is a file expression and list is either a list of numeric array names separated by commas or a list of string array names separated by commas.

Semantics:

The mat-write-file statement causes all of the elements of an array with subscripts that are not 0 to be written into the random file with file number n. The type of file n must be the same as the type of the arrays in the list. The file number cannot be 0.

The mat-write-file statement writes values into the random file starting with the current position of the file pointer. The file pointer is incremented by 1 after each data value is written.

```
100 mat write #3: a,b
200 mat write #4: c$
```

## SECTION 7

## SAMPLE PROGRAMS

This section discusses a number of sample programs that illustrate various features of BASIC. All of these programs have been executed on Multics.

### EXAMPLE 1

This program prints a report giving the gasoline mileage for a series of trips. It illustrates how data can be contained in the program body, how simple computations can be performed, and how the standard print formats are used.

Line 110 prints a heading for the report. Since the strings are separated by commas, they are printed in the standard zones of 15 columns.

Line 120 reads the starting odometer value. The data values are provided in data statements located in the program body. The first numeric value read comes from line 260. Since the dates in lines 270 to 330 start with a number, quotes are required. All of the data statements are skipped after line 250 is executed.

The main part of the program is contained in lines 140 to 240. An odometer value is read; if it is 0 the loop is terminated. If the odometer value is not 0, a date string and the amount of gasoline used are read. Note that the order of string data values is independent of the numeric data values. The first time line 190 is executed, variable m1 has its initial value of 0. Line 210 prints the statistics for a single trip.

Line 250 prints the total mileage, gasoline used, and average miles per gallon. Two consecutive commas cause a zone to be skipped.

Here is the program for example 1:

```
100 rem Total Miles Per Gallon
110 print "Date", "Odometer", "Trip", "Gallons", "  MPG"
120 read k
130 let k1 = k
140 read n
150 if n = 0 then 250
160 read d$
170 read g
180 let m = n - k
190 let m1 = m1 + m
200 let a = m/g
210 print d$,n,m,g,a
220 let k = n
230 let g1 = g1 + g
240 goto 140
250 print "TOTAL",,m1,g1,(k-k1)/g1
260 data 3332
270 data "1/10/73", 3553, 14.8
280 data "1/17/73", 3801, 17.4
290 data "1/20/73", 3926, 7.2
300 data "1/27/73", 4091, 11.3
310 data "2/3/73", 4275, 10.9
320 data "2/9/73", 4460, 9.8
330 data "2/15/73", 4664, 12.3
340 data 0
350 end
```

Here is the output for example 1:

```
Date          Odometer      Trip          Gallons       MPG
1/10/73       3553          221           14.8          14.9324
117/73        3801          248           17.4          14.2529
1/20/73       3926          125           7.2           17.3611
1/27/73       4091          165           11.3          14.6018
2/3/73        4275          184           10.9          16.8807
2/9/73        4460          185           9.8           18.8776
2/15/73       4664          204           12.3          16.5854
TOTAL                       1332          83.7          15.914
```

EXAMPLE 2

This is the program of example 1 modified so that data is read from a file instead of from data statements in the program body. This change permits the program to be more conveniently used by a number of people; each user only has to prepare an input file.

Line 110 associates the file name "gas_data" with file number 1. Line 150 checks to see if the end of the input file has been reached; execution continues with line 160 if there are more data in the file. The order of the references read in line 160 must correspond to the types of data in the input file. Quotes are not required around the dates in the input file because the BASIC runtime system knows that strings are required.

The output format has been improved in example 2. The default format used in example 1 printed the gasoline mileage with four fractional places. The function defined in line 250 returns the value of its input parameter rounded to one decimal place; this function is used to control the output format in lines 140 and 240. Note that the function body is ignored when line 250 is encountered after line 240.

Here is the program for example 2:

```
100 rem Total Miles Per Gallon
110 file #1: "gas_data"
120 input #1: k
130 let k1 = k
140 print "Date", "Odometer", "Trip", "Gallons", "  MPG"
150 if end #1 then 240
160 input #1: d$,n,g
170 let m = n - k
180 let m1 = m1 + m
190 let a = m/g
200 print d$,n,m,g,fnr(a)
210 let k = n
220 let g1 = g1 + g
230 goto 150
240 print "TOTAL",,m1,g1,fnr((k-k1)/g1)
250 def fnr(x) = int(x*10 + .5)/10
260 end
```

Here is the input data file for example 2:

```
3332
1/10/73, 3553, 14.8
1/17/73, 3801, 17.4
1/20/73, 3926, 7.2
1/27/73, 4091, 11.3
2/3/73, 4275, 10.9
2/9/73, 4460, 9.8
2/15/73, 4664, 12.3
```

Here is the output for example 2:

```
Date              Odometer          Trip             Gallons            MPG
1/10/73            3553              221              14.8              14.9
1/17/73            3801              248              17.4              14.3
1/20/73            3926              125              7.2               17.4
1/27/73            4091              165              11.3              14.6
2/3/73             4275              184              10.9              16.9
2/9/73             4460              185              9.8               18.9
2/15/73            4664              204              12.3              16.6
TOTAL                                1332             83.7              15.9
```

EXAMPLE 3

This is the program of example 2 modified to use the print-using statement. Lines 200 and 240 contain "pictures" of the desired output format. The print-using statement permits the fields to be printed closer together and allows decimal points to be lined up.

Here is the program for example 3:

```
 100 rem Total Miles Per Gallon
 110 file #1: "gas_data"
 120 input #1: k
 130 let k1 = k
 140 f1$="<######  -#####  -###   -##.#      -##.#",d$,n,m,g,a
 150 f2$="TOTAL            -###   -##.#      -##.#",m1,g1,(k-k1)/g1
 160 print "Date    Odometer  Trip  Gallons  Miles / Gallon"
 170 if end #1 then 240
 180 input #1: d$,n,g
 190 let m = n - k
 200 let m1 = m1 + m
 210 let a = m/g
 220 print using f1$
 230 let k = n
 240 let g1 = g1 + g
 250 goto 150
 260 print using f2$
 270 end
```

Here is the output from example 3 using the input file from example 2:

```
Date     Odometer  Trip  Gallons  Miles / Gallon
1/10/73    3553     221    14.8       14.9
1/17/73    3801     248    17.4       14.3
1/20/73    3926     125     7.2       17.4
1/27/73    4091     165    11.3       14.6
2/3/73     4275     184    10.9       16.9
2/9/73     4460     185     9.8       18.9
2/15/73    4664     204    12.3       16.6
TOTAL              1332    83.7       15.9
```

## EXAMPLE 4

This program shows how the gosub- and return statements can be used to execute a block of statements from several points in the program. Lines 120 to 170 evaluate the greatest common divisor of three integers by using the relationship

gcd(x,y,z) = gcd(gcd(x,y),z)

Lines 230 to 300 evaluate the greatest common divisor of two integers using the Euclidian algorithm. Note that it is possible to transfer control to a rem statement and that the same statement can be the target of both goto- and gosub statements. The first time line 300 is executed, control returns to line 150; the second time line 300 is executed, control returns to line 180. The message

```
Out of data in 110
```

is printed and program execution is terminated when all the data values have been read.

Here is the program for example 4:

```
100 print "A", "B", "C", "GCD"
110 read a,b,c
120 let x = a
130 let y = b
140 gosub 230
150 let x = g
160 let y = c
170 gosub 230
180 print a,b,c,g
190 goto 110
200 data 6,9,12
210 data 38456, 64872, 98765
220 data 32, 384, 72
230 rem
240 let r = mod(x,y)
250 if r = 0 then 290
260 let x = y
270 let y = r
280 goto 230
290 let g = y
300 return
310 end
```

Here is the output from example 4:

```
A              B              C              GCD
 6              9              12             3
 38456          64872          98765          1
 32             384            72             8
Out of data in 110
```

## EXAMPLE 5

This is the program of example 4 modified to compute the greatest common divisor of two integers by means of a multiple line function. The function is defined in lines 150 to 220. Variable r is local to the function body. The assignments to x and y in lines 180 and 190 do not alter the values of the function arguments.

Here is the program for example 5:

```
100 print "A", "B", "C", "GCD"
110 read a,b,c
120 print a,b,c,fng(fng(a,b),c)
130 goto 110
140 rem
150 def fng(x,y)r
160 let r = mod(x,y)
170 if r = 0 then 210
180 let x = y
190 let y = r
200 goto 160
210 let fng = y
220 fnend
230 data 6,9,12
240 data 38456, 64872, 98765
250 data 32, 384, 72
260 end
```

Here is the output of example 5:

```
A              B              C              GCD
 6              9              12              3
 38456          64872          98765           1
 32             384            72              8
Out of data in 110
```

## EXAMPLE 6

This program illustrates the use of a recursive function. The relations

f(0) = 1 f(n) = n * f(n-1)

are used to define the factorial function. The use of fnf to the left of the equal sign in line 180 is a reference to the function value; the use of fnf to the right of the equal sign in line 180 is an invocation of the function.

Line 100 prints the message "n = " without a newline character so that the message, the input prompt, and the reply can all appear on the same line of output. Line 120 prints the three values with no extra intervening spaces. (One space is automatically put at the end of the value.) The program repeatedly requests a value of n from the user. A message is printed and execution is terminated when the user responds "stop" to the input prompt.

Here is the program for example 6:

```
100 print "n = ";
110 input n
120 print n;"  =";fnf(n)
130 goto 100
140 def fnf(x)
150 if x <> 0 then 180
160 let fnf = 1
170 goto 190
180 let fnf = x * fnf(x-1)
190 fnend
200 end
```

Here is the output for example 6:

```
n = ? 4
 4    = 24
n = ? 5
 5    = 120
n = ? 10
 10    = 3628800
n = ? 15
 15    = 1.30767 E+12
n = ? stop
Program halted in 110
```

<u>EXAMPLE 7</u>

This program uses the array processing capabilities of BASIC to compute the total dollar value of three products sold by five salesmen. Vector element p(m) is the price of the mth product. Matrix element p(m,n) is the total number of the mth product sold by the nth salesman.

Line 110 reads three values into the p vector; line 120 reads 15 values into the s matrix. Since neither array was redimensioned, the current bounds are the same as the original bounds. Lines 130 to 190 compute and print the total sales for each salesman. The dollar sign in the numeric field in line 180 floats up against the leftmost digit printed in the field.

Here is the program for example 7:

```
100 dim s(3,5),p(3)
110 mat read p
120 mat read s
130 for n = 1 to 5
140 let s = 0
150 for m = 1 to 3
160 let s = s + p(m) * s(m,n)
170 next m
180 print using "Total sales for salesman-# $-###.##", n, s
190 next n
200 data 1.25, 4.30, 2.50
210 data 40, 20, 2, 29, 42
220 data 10, 16, 3, 21, 8
230 data 35, 47, 29, 16, 33
240 end
```

Here is the output from example 7:

```
Total sales for salesman 1  $180.50
Total sales for salesman 2  $211.30
Total sales for salesman 3   $87.90
Total sales for salesman 4  $166.55
Total sales for salesman 5  $169.40
```

## EXAMPLE 8

This is the program of example 7 modified to handle an arbitrary number of products and an arbitrary number of salesmen. Input now comes from a file whose name is requested in line 110. The first two values in the file are the number of products and the number of salesmen, which are read in line 140. The product vector and sales matrix are redimensioned with the appropriate bounds in the input-statment in line 150. The total sales vector t is generated by a matrix multiplication operation in line 160.

Here is the program for example 8:

```
100 dim s(20,50), p(20), t(50)
110 print "Data file name";
120 input f$
130 file #1: f$
140 input #1: p,s
150 mat input #1: p(p), s(p,s)
160 mat t = p * s
170 for n = 1 to s
180 print using "Total sales for salesman-# $-###.##",n,t(n)
190 next n
200 end
```

Here is the data file (named sales_data) for example 8:

```
3, 5
1.25, 4.30, 2.50
40, 20, 2, 29, 42
10, 16, 3, 21, 8
35, 47, 29, 16, 33
```

Here is the output for example 8:

```
Data file name? sales_data
Total sales for salesman 1  $180.50
Total sales for salesman 2  $211.30
Total sales for salesman 3   $87.90
Total sales for salesman 4  $166.55
Total sales for salesman 5  $169.40
```

## EXAMPLE 9

This program generates random sentences. It is given a list of word keys, a set of sentence patterns that use these keys to encode a sentence, and sets of words of given word types. It picks a sentence pattern at random and picks a word at random from each word class to make up the sentence.

Line 100 causes a different sequence of pseudorandom numbers to be generated each time the program is run.

Lines 170 to 240 read in the file "parts_list". Each line of the parts list file consists of a single character key and the name of a file containing words of that type. Line 190 reads the character key into the next element of the p$ vector and gets the file name associated with it. The entire file of words of the specified type is then read into the columns of the next row of the w$ matrix. Note that the same file number is used for each words file; each time line 200 is executed, the file previously assigned to the file number is closed.

Lines 300 to 330 read the sentence patterns from the "sentence_list" file. Line 380 picks a pattern at random from the vector of saved patterns. The pattern is printed by line 390. Line 450 initializes the sentence to the null string.

Lines 460 to 520 inspect each character of the chosen pattern. If the next character from the pattern matches one of the keys, control goes to line 580. The pattern is incorrect if the character matches none of the keys.

Lines 580 to 610 pick a word at random from the class of words corresponding to the key. Up to five tries are made to avoid putting the same word in the sentence more than once. Line 650 appends the new word to the sentence and follows it by a blank. The sentence gets printed by line 670.

Here is the program for example 9:

```
100        randomize
110 '
120 ' read in parts list, format of each line is
130 '      k,file
140 ' where k is a 1 character key and file is the
150 ' name of a file of words corresponding to the key.
160 '
170        file #1: "parts_list"
180        let p = p + 1
190        input #1: p$(p), a$
200        file #2: a$
210        linput #2: w$(p,w(p))
220        let w(p) = w(p) + 1
230        if more #2 then 210
240        if more #1 then 180
250 '
260 ' at this point, the keys are in the p$ array and
270 ' all the words of type i are in w$(i,1) ... w$(i,w(i))
280 ' now read in list of sentence patterns
290 '
300        file #1: "sentence_list"
310        linput #1: s$(s)
320        let s = s + 1
330        if more #1 then 310
340 '
350 ' this main loop, pick sentence pattern at random
360 ' and print pattern at start of line
370 '
380        let s$ = s$(int(s*rnd))
390        print s$;":";tab(10);
400 '
410 ' initialize sentence string;  replace each key
420 ' character in the pattern by a word chosen at
430 ' random from list of words of given type
440 '
450        let x$ = ""
460        for i = 1 to len(s$)
470             let a$ = seg$(s$,i,i)
480             for j = 1 to p
490                  if a$ = p$(j) then 580
500                  next j
510             print "Illegal sentence pattern."
520             goto 380
530 '
540 ' this is word type j, pick word from class j
550 ' at random try (up to 5 times) to avoid putting same
560 ' word in sentence more than once
570 '
580             for t = 1 to 5
590                  let b$ = w$(j,w(j)*rnd)
```

```
600                 if pos(x$,b$,1) = 0 then 650
610                 next t
620 '
630 ' add word to sentence
640 '
650             let x$ = x$ & b$ & " "
660             next i
670         print x$
680         goto 380
690         dim w$(20,100),w(20),s$(40),p(20)
700         end
```

Here is the parts_list file for example 9:

```
n,noun_list
d,adjective_list
v,transitive_verb_list
r,article_list
b,adverb_list
p,pronoun_list
x,aux_list
t,preposition_list
w,intransitive_verb_list
i,interjection_list
```

Here is the first part of the sentence_list file:

```
rnixd
rnixrn
rnixrdn
bpvrn
brnvrn
pbvrn
pbvrn
pbwtrn
pvb
pvrn
pwb
pwtrn
```

Here are some of the sentences that were generated:

|  |  |
|---|---|
| pwb: | he swims swiftly |
| pvrn: | it finds a paper |
| pwtrn: | it goes near the boy |
| pvrn: | it produces the computer |
| pbvrn: | he quickly obtains the policeman |
| rnixd: | the refrigerator , as I said before, is white |
| pwtrn: | she walks far from the home |
| pxtrn: | it was far from the fireman |
| pvrn: | he gets the fork |
| pbvrn: | she busily eats the tomato |
| pbvrn: | she busily loses a bottle |
| rdnwb: | the short fish swims slowly |
| rdnvrn: | a dumb fireman avoids the secretary |
| bpvrn: | happily he gets the manager |
| pwtrn: | he runs around the fish |
| pwb: | he walks happily |
| pwtrn: | he goes to a manager |
| pwb: | she walks happily |
| pbvrn: | he busily writes a woman |
| pxd: | he was full |
| pxd: | he will be sad |
| pvrn: | she writes a boy |
| pvrn: | he avoids the fork |
| pxtrn: | she is far from the tree |
| pvrn: | he obtains the book |

## EXAMPLE 10

This program shows how a BASIC program can be called by the Multics command processor. The program converts a terminal format file into a random string file where each line of the input file becomes an entry in the output file.

Line 100 obtains the number of command arguments of the program. A message is printed and execution stops if less than two arguments are provided.

Lines 140 to 170 set the value that is used for the random file margin. If only two arguments are given, a default margin of 32 is used. If three (or more) arguments are present, the third argument is obtained and converted to a numeric value by line 170.

Lines 180 and 190 get the names of the input and output files. An error message is printed if the names are the same.

The real work of the program is done in lines 230 to 300. Line 250 erases any previous contents of the output file. Line 260 sets the margin on the output file. Line 280 reads an entire line from the output file; line 290 writes the string into the output file.

Here is the program for example 10:

```
100 let n = cnt
110 if n >= 2 then 140
120 print "Not enough arguments supplied."
130 stop
140 if n >= 3 then 170
150 let m = 32
160 goto 180
170 let m = val(arg$(3))
180 let a$ = arg$(1)
190 let b$ = arg$(2)
200 if a$ <> b$ then 230
210 print "Cannot use same name as infile & outfile"
220 stop
230 file #1: a$
240 file #2: b$
250 scratch #2
260 margin #2: m
270 if end #1 then 310
280 linput #1: c$
290 write #2: c$
300 goto 270
310 end
```

## EXAMPLE 11

This is the sentence generator of example 9 modified to pick words at random from a set of random access files. The program of example 10 is used to convert the terminal format word files used by example 9 into the random string files used by this example.

Lines 140 to 180 read in the sentence keys and the associated random file names. Instead of storing the keys in a vector, this program forms a string consisting of all the character keys concatenated together. Lines 220 to 250 read in the sentence list.

Line 290 generates a random sentence pattern that is processed by lines 340 to 540. Line 350 checks if the next character in the pattern is one of the keys that are recognized. Control goes to line 420 if the character is valid.

Line 420 opens the file whose name is associated with the current word key. Line 440 generates an integer k that ranges from 0 to N-1 where N is the length of the file. Line 450 positions the file pointer so that line 460 reads the selected word. As in example 9, five tries are made to pick a word that does not already occur in the sentence.

This program can deal with a much larger vocabulary than the program of example 9 because it has to store only the word it is using rather than all the words it can ever use. However, for small word files, the program of example 9 runs faster.

Here is the program for example 11:

```
100       randomize
110 '
120 ' read in parts list
130 '
140       file #1: "parts_file"
150       let p = p + 1
160       input #1: x$, f$(p)
170       let p$ = p$ & x$
180       if more #1 then 150
190 '
200 ' read in sentence patterns
210 '
220       file #1: "sentence_list"
230       linput #1: s$(s)
240       let s = s + 1
250       if more #1 then 230
260 '
270 ' pick random pattern
280 '
290       let s$ = s$(int(s*rnd))
300       let x$ = ""
310 '
320 ' process each word type in pattern
330 '
340       for i = 1 to len(s$)
350            let j = pos(p$,seg$(s$,i,i),1)
360            if j <> 0 then 420
370            print "Illegal sentence pattern ";s$
380            goto 290
390 '
400 ' pick word at random from file j
410 '
420            file #1: f$(j)
430            for t = 1 to 5
440                 let k = int(lof(#1) * rnd)
450                 reset #1: k
460                 read #1: b$
470                 if pos(x$,b$,1) = 0 then 520
480                 next t
490 '
500 ' add word to sentence
510 '
520            let x$ = x$ & b$ & " "
530            next i
540       print x$
550       goto 290
560       dim s$(50)
570       end
```

Here are some sentences that were generated:

```
she was tall
a sad tall toy travels dumbly
the dog happily shakes the waiter
the crowded small car runs slowly
the secretary is near the desk
he writes a letter
```

## EXAMPLE 12

This program is a "desk calculator" that evaluates simple expressions consisting of constants, BASIC operators, and BASIC functions. It does this by turning the expression into a small BASIC program to print the value of the expression; the BASIC compiler is then called to compile and execute the small program.

Line 100 defines the name of the intermediate BASIC source file. Line 110 requests that an expression be provided; a response of "stop" causes the calculator to stop. Line 140 calls subprogram "write" to write the expression into the file; this organization takes advantage of the fact that a file opened by a subprogram is closed when the subprogram returns. Line 150 calls the BASIC compiler, located in a separate segment, to compile and execute the program in the temporary file.

Here is the "calculator"

```
100 let b$ = "TEMP.basic"
110 print "Input expression";
120 linput a$
130 if a$ = "stop" then 170
140 call "write": a$, b$
150 call "basic": b$
160 goto 110
170 end
180 sub "write": f$, g$
190 file #1: g$
200 scratch #1
210 print #1: "100 print ";f$
220 print #1: "200 end"
230 subend
```

Here are some expressions it evaluated:

```
Input expression? sin(.47)^2 + cos(.47)^2
 1
Input expression? 1-sin(.33)*sqr(.731e12)
-277051
Input expression? 1+2+3+4+5+6+7+8+9;sqr(25);0
 45  5  0
Input expression? stop
```

# APPENDIX A

## ASCII CHARACTER SET

| Graphic | Octal Value | Decimal Value | ASC Function Abbrev | Comments |
|---|---|---|---|---|
| | 0 | 0 | nul | Null |
| | 1 | 1 | soh | Start of heading |
| | 2 | 2 | stx | Start of text |
| | 3 | 3 | etx | End of text |
| | 4 | 4 | eot | End of transmission |
| | 5 | 5 | enq | Enquiry |
| | 6 | 6 | ack | Acknowledge |
| | 7 | 7 | bel | Bell |
| | 10 | 8 | bs | Backspace |
| | 11 | 9 | ht | Horizontal tab |
| | 12 | 10 | lf,nl | Line feed, Multics newline |
| | 13 | 11 | vt | Vertical tab |
| | 14 | 12 | ff | Form feed |
| | 15 | 13 | cr | Carriage return |
| | 16 | 14 | so | Shift out |
| | 17 | 15 | si | Shift in |
| | 20 | 16 | dle | Data link escape |
| | 21 | 17 | dc1 | Device control 1 |
| | 22 | 18 | dc2 | Device control 2 |
| | 23 | 19 | dc3 | Device control 3 |
| | 24 | 20 | dc4 | Device control 4 |
| | 25 | 21 | nak | Negative acknowledge |
| | 26 | 22 | syn | Synchronous idle |
| | 27 | 23 | etb | End of transmission block |
| | 30 | 24 | can | Cancel |
| | 31 | 25 | em | End of medium |
| | 32 | 26 | sub | Substitute |
| | 33 | 27 | esc | Escape |
| | 34 | 28 | fs | File separator |
| | 35 | 29 | gs | Group separator |
| | 36 | 30 | rs | Record separator |
| | 37 | 31 | us | Unit separator |
| | 40 | 32 | sp | Space |
| ! | 41 | 33 | ! | Exclamation point |

| Graphic | Octal Value | Decimal Value | ASC Function Abbrev | Comments |
|---|---|---|---|---|
| " | 42 | 34 | qt,quo | Quotation mark |
| # | 43 | 35 | # | Number sign |
| $ | 44 | 36 | $ | Dollar sign |
| % | 45 | 37 | % | Percent sign |
| & | 46 | 38 | & | Ampersand |
| ' | 47 | 39 | apo | Acute accent, apostrophe |
| ( | 50 | 40 | ( | Left parenthesis |
| ) | 51 | 41 | ) | Right parenthesis |
| * | 52 | 42 | * | Asterisk |
| + | 53 | 43 | + | Plus |
| , | 54 | 44 | , | Comma |
| - | 55 | 45 | - | Minus |
| . | 56 | 46 | . | Period |
| / | 57 | 47 | / | Slash |
| 0 | 60 | 48 | 0 | Zero |
| 1 | 61 | 49 | 1 | One |
| 2 | 62 | 50 | 2 | Two |
| 3 | 63 | 51 | 3 | Three |
| 4 | 64 | 52 | 4 | Four |
| 5 | 65 | 53 | 5 | Five |
| 6 | 66 | 54 | 6 | Six |
| 7 | 67 | 55 | 7 | Seven |
| 8 | 70 | 56 | 8 | Eight |
| 9 | 71 | 57 | 9 | Nine |
| : | 72 | 58 | : | Colon |
| ; | 73 | 59 | ; | Semicolon |
| < | 74 | 60 | < | Less than |
| = | 75 | 61 | = | Equals |
| > | 76 | 62 | > | Greater than |
| ? | 77 | 63 | ? | Question mark |
| @ | 100 | 64 | @ | Commercial at |
| A | 101 | 65 | uca | Uppercase A |
| B | 102 | 66 | ucb | Uppercase B |
| C | 103 | 67 | ucc | Uppercase C |
| D | 104 | 68 | ucd | Uppercase D |
| E | 105 | 69 | uce | Uppercase E |
| F | 106 | 70 | ucf | Uppercase F |
| G | 107 | 71 | ucg | Uppercase G |
| H | 110 | 72 | uch | Uppercase H |
| I | 111 | 73 | uci | Uppercase I |
| J | 112 | 74 | ucj | Uppercase J |
| K | 113 | 75 | uck | Uppercase K |
| L | 114 | 76 | ucl | Uppercase L |
| M | 115 | 77 | ucm | Uppercase M |
| N | 116 | 78 | ucn | Uppercase N |
| O | 117 | 79 | uco | Uppercase O |
| P | 120 | 80 | ucp | Uppercase P |

| Graphic | Octal Value | Decimal Value | ASC Function Abbrev | Comments |
|---|---|---|---|---|
| Q | 121 | 81 | ucq | Uppercase Q |
| R | 122 | 82 | ucr | Uppercase R |
| S | 123 | 83 | ucs | Uppercase S |
| T | 124 | 84 | uct | Uppercase T |
| U | 125 | 85 | ucu | Uppercase U |
| V | 126 | 86 | ucv | Uppercase V |
| W | 127 | 87 | ucw | Uppercase W |
| X | 130 | 88 | ucx | Uppercase X |
| Y | 131 | 89 | ucy | Uppercase Y |
| Z | 132 | 90 | ucz | Uppercase Z |
| [ | 133 | 91 | [ | Left bracket |
| \ | 134 | 92 | \ | Reverse slash |
| ] | 135 | 93 | ] | Right bracket |
| ^ | 136 | 94 | ^ | Circumflex, up arrow |
|  | 137 | 95 | _,und,bkr | Underscore, back arrow |
| ` | 140 | 96 | ` | Grave accent |
| a | 141 | 97 | a,lca | Lowercase a |
| b | 142 | 98 | b,lcb | Lowercase b |
| c | 143 | 99 | c,lcc | Lowercase c |
| d | 144 | 100 | d,lcd | Lowercase d |
| e | 145 | 101 | e,lce | Lowercase e |
| f | 146 | 102 | f,lcf | Lowercase f |
| g | 147 | 103 | g,lcg | Lowercase g |
| h | 150 | 104 | h,lch | Lowercase h |
| i | 151 | 105 | i,lci | Lowercase i |
| j | 152 | 106 | j,lcj | Lowercase j |
| k | 153 | 107 | k,lck | Lowercase k |
| l | 154 | 108 | l,lcl | Lowercase l |
| m | 155 | 109 | m,lcm | Lowercase m |
| n | 156 | 110 | n,lcn | Lowercase n |
| o | 157 | 111 | o,lco | Lowercase o |
| p | 160 | 112 | p,lcp | Lowercase p |
| q | 161 | 113 | q,lcq | Lowercase q |
| r | 162 | 114 | r,lcr | Lowercase r |
| s | 163 | 115 | s,lcs | Lowercase s |
| t | 164 | 116 | t,lct | Lowercase t |
| u | 165 | 117 | u,lcu | Lowercase u |
| v | 166 | 118 | v,lcv | Lowercase v |
| w | 167 | 119 | w,lcw | Lowercase w |
| x | 170 | 120 | x,lcx | Lowercase x |
| y | 171 | 121 | y,lcy | Lowercase y |
| z | 172 | 122 | z,lcz | Lowercase z |
| { | 173 | 123 | {,lbr | Left brace |
| ¦ | 174 | 124 | ¦,vln | Vertical line |
| } | 175 | 125 | },rbr | Right brace |
| ~ | 176 | 126 | ~,til | Tilde |
|  | 177 | 127 | del | Delete |

# APPENDIX B

# COMPATIBILITY WITH NON-BASIC PROGRAMS

A BASIC program can call programs written in FORTRAN or PL/I and can be called by programs written in these languages. The only allowable argument types for a call involving a non-BASIC program are numeric scalars, numeric arrays, and string scalars. String arrays, functions, and files cannot be passed as arguments or received as parameters by BASIC programs when a program written in another language is involved.

## CALLS BETWEEN BASIC AND PL/I

The following table gives the correspondence between the data types of BASIC and the equivalent data types of PL/I.

| BASIC | PL/I |
|---|---|
| numeric scalar | float bin(27) |
| numeric vector M | float bin(27) dim(0:M) |
| numeric matrix M x N | float bin(27) dim(0:M,0:N) |
| string scalar | char(*) |

All arguments are passed by reference, except when a PL/I character string is passed to a BASIC program; then the value of the PL/I string expression is copied by BASIC at entry and is written back at exit. When a BASIC character string is passed to a PL/I program, the current length is used. Thus, if the PL/I program is to return a value, the BASIC program must first initialize the string to a value with an appropriate length. When a PL/I array, which must be connected, is passed to a BASIC program, the lower bound of each dimension is adjusted to equal 0; thus an array dimensioned P:Q in the PL/I program is seen as 0:Q-P by the BASIC program.

## CALLS BETWEEN BASIC AND FORTRAN

The following table gives the correspondence between the data types of BASIC and the equivalent data types of FORTRAN.

| BASIC | FORTRAN |
|---|---|
| numeric scalar | real |
| numeric vector N | real, dimension N1 |
| string scalar | character*K where K is length of BASIC string |

Passing arrays between BASIC and FORTRAN programs is complicated by the fact that BASIC stores arrays in row-major order with a lower bound of 0 for each dimension while FORTRAN stores arrays in column-major order with a lower bound of 1 for each dimension. For these reasons, two-dimensional arrays cannot be passed between BASIC and FORTRAN programs and the bounds and subscripts of one-dimensional arrays must be adjusted; when a vector is passed, the bounds and subscripts used by the FORTRAN program must be 1 greater than the bounds and subscripts used by the BASIC program.

# APPENDIX C

# BASIC FILE ATTACHMENTS

This appendix lists the I/O switch attachments that can be specified in a BASIC file name.

## FILES IN THE STORAGE SYSTEM

The attach description must be of the form

```
vfile_ f
```

where f is an absolute or relative pathname that identifies a file.

## FILES ON TAPE

The attach description must be of the form

```
record_stream_ -target ntape_ r -raw -write
```

where r is a string identifying the reel to be read or written. The string r should end with the sequence ",7track" or ",9track" to indicate the type of tape to be read or written. If neither of these endings are present, ",9track" is assumed.

The -write control argument causes the reel to be mounted with a write-permit ring. This control argument is required if the program contains print-statements or scratch-statements that access the file.

The -raw control argument is required; it means that each line in the file corresponds to a single physical tape record.

## TERMINAL INPUT/OUTPUT

The attach description must be of the form

tty_ d

where d is the string, obtainable from the print_attach_table (pat) command or user active function, that identifies the terminal device assigned to the I/O switch name user_i/o in the user's process.

## SYNONYM ATTACHMENTS

The attach description must be of the form

syn_ n

where n is the name of an I/O switch through which all operations on this switch are to be directed. Such a switch must exist at the time the switch is opened, although it need not exist when the switch is attached. The I/O switch whose name is n can itself be attached as a synonym for another I/O switch. The I/O switch that is the final destination of the synonym attachment must be attached to a file or device and must specify an I/O module.

For more information on the Multics Input/Output System, refer to the MPM Reference Guide under "Input and Output Facilities."

# APPENDIX D

# EXTENDED PRECISION

BASIC is available in extended as well as single precision, which is the default. Programs compiled in extended precision mode do all numeric processing in double precision. These programs should not call or be called by single precision programs because numeric arguments, including numeric files, are not compatible.

To compile in extended precision mode, type the Multics command use_ep_basic with no arguments. All BASIC programs compiled after that will use double precision arithmetic. This effect lasts only for the life of the process or run unit or until use_sp_basic is typed, which returns the compiler to single precision mode. Note that these commands affect only the compiler; programs of either precision can be run at any time.

To convert numeric files from single to double precision or vice versa, use the convert_numeric_file command described below.

## CONVERT_NUMERIC_FILE

The convert_numeric_file command converts numeric files used by BASIC programs from single to double precision and vice versa using PL/I conversion rules.

### SYNTAX AS A COMMAND

```
convert_numeric_file OLD_PATH NEW_PATH -CONTROL_ARGS
```

### ARGUMENTS

OLD_PATH
: is the pathname of the file to be converted.

NEW_PATH
: is the pathname of the converted file.

### CONTROL ARGUMENTS

-double_precision,
-dp
: converts from single to double precision; this is the default.

-single_precision,
-sp
: converts from double to single precision.

# INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

| | | | |
|---|---|---|---|
| TITLE | LEVEL 68<br>MULTICS BASIC MANUAL | ORDER NO. | AM82-01 |
| | | DATED | FEBRUARY 1981 |

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____________________

CUT ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

**Honeywell**