MULTICS BASIC MANUAL
ADDENDUM A

SUBJECT

Additions and Changes to the Manual

SPECIAL INSTRUCTIONS

Refer to the Preface for "Significant Changes."

This is the first addendum to AM82, Revision 1, February 1981. Throughout the document, change bars are used to indicate technical changes and additions; asterisks denote deletions. These changes will be incorporated in the next revision of this manual.

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Note: Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

AM82-01A

December 1984

41627
7.5C1184
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
title page, preface	title page, preface
iii through vii, blank	iii through vi
1-1, 1-2	1-1, 1-2 1-2.1, blank
3-1 through 3-7, blank	3-1 through 3-7, blank
4-3, 4-4	4-3, blank 4-3.1, 4-4
4-7, blank	4-7, blank
5-3, 5-4	5-3, 5-4
5-11, 5-12	5-11, 5-12 5-12.1, blank
5-17, 5-18	5-17, 5-18
5-27, 5-28	5-27, 5-28 5-28.1, blank
5-31, 5-32	5-31, 5-32
C-1, C-2	C-1, C-2
i-1 through i-9, blank	i-1 through i-5, blank

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 Commands and Active Functions manual, Order No. AG92. Also, this manual does not attempt to provide the reader with basic knowledge of the Multics system. The reader is referred to the New Users' Introduction to Multics -- Part I manual, Order No. CH24 and to the New Users' Introduction to Multics -- Part II manual, Order No. CH25 for an introduction to Multics. In addition, the reader is referred to the Multics FAST Subsystem Reference Guide Order No. AU25 describing the time-sharing facility supporting BASIC and FORTRAN program development.

Addendum A contains documentation support for new enhancements as described below:

The following string functions are new:

```
mid$(a$,i,j)
left$(a$,i)
right$(a$,i)
```

The string function 'pos' can have an optional number of arguments.

'+' can be used for concatenation along with the '&'.

BASIC programs can be written without line numbers.

CONTENTS

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 1 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.

Line numbering in Multics BASIC is optional. Line numbers can be used as labels for statements that require labels such as the goto or gosub statements. Statements that do not have line numbers must be preceded by a backslash (\). The following example illustrates a small BASIC program that does not employ line numbers.

Example

```
\s = 0
\for i = 1 to 100
\    s = s + i
\next i
\print "The sum is ";s
\end
```

If you choose to use line numbers for your programs or you use some line numbers for statement labels, each line number must be greater than the one preceding it.

Other subsystems (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.

This page intentionally left blank.

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 either of the two concatenation operators & or +. These operators combine 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)
a$ + b$ & left$(c$,3)
```

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 to five 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.

Function	Description
left$(a$,i)	The substring of a$ that consists of the first i' characters, where i' = min (i, len(a$)). If i < 0 a zero-length string is returned.
len(a$)	The number of characters in the string a$.
log(x)	The logarithm of x to the base e.
max(x1,...,xn)	The maximum of n numeric values. This function allows an arbitrary number of arguments.
mid$(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$)-i'+1),0). This function is equivalent to sst$.
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$, starting at or after position i in a$, if the last argument is supplied, or position 1 in a$, if the last argument is omitted.
right$(a$,i)	The substring of a$ that consists of the last i' characters, where i' = min (i,len(a$)). If i < 0 a zero-length string is returned.
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.

Function	Description
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$)-i'+1),0). This function is equivalent to mid$.
str$(x)	The string that is the decimal representation of the numeric value of x. The conversion follows the rules for printed output. (See Section 5.)
tan(x)	The tangent of x, where x is in radians.
tim	The elapsed running time of the program in seconds. This value is determined from the microsecond clock used by the Multics system.
tst(a$)	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. The maximum number of invocations is dependent on stack space. In the simplest case, the maximum number of active invocations is 51. However, if local variables are used and/or if gosubs or other multiple line functions are invoked, this number is decreased.

This page intentionally left blank.

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 Multics Programmer's Reference Manual, Order No. AG91) 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.

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 Multics Programmer's Reference Manual, Order No. AG91, for further information on special characters.

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. Functions with a variable number of arguments, such as max, min, and pos cannot be passed as function arguments.

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 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.

In general, 255 gosub statements can be executed before a return statement; however, the number may be less if multiple-line functions are also executed. 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.

This page intentionally left blank.

```
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:

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 "$" is not immediately followed by a "+" or "-", "-" is assumed to be inserted before the "#" following the "$".

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

This page intentionally left blank.

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

Field	Internal Value	External Form
-##	2	ØØ2
-##	-23	-23
-##	476	476
-##	-476	***
+##	23	+23
+##	-23	-23
+##	476	***
+##	0	Ø+0
-##.##	17.479	Ø17.48
-##.##	1.7479	ØØ1.7
-##.##	.17479	ØØ0.17
-##.##	-.172	Ø-0.17
-.##	0.23	0.23
-.##	-0.23	-.23
-	7	7
-	-7	*
$-#,###.00	18.43	ØØØØ$18.00
$-#,###.00	-1234	$-1,234.00
-##.##^^^^^	123.4	Ø12.34 E+1Ø
-##.##^^^^^	-1.234e14	-12.34 E+13
-##.##^^^^^	0	Ø00.00 E+0Ø
$###.##	25.4	bb$25.40
$#.#	-7	$-7.0

Where "b" is assumed to be the blank character.

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 (Ø indicates a single blank):

Field	Internal Value	External Form
<########	alpha	alphaØØØØ
>########	beta	ØØØØbeta
>########	betaØ	ØØØbetaØ
<##	alpha	alp
>##	alpha	pha
<1#2#3#4#	alpha	a1¦2p3h4a

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.

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 Multics Programmer's Reference Manual, Order No. AG91.

INDEX

A

B

C

P

R

S

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS BASIC MANUAL
ADDENDUM A

ORDER NO. AM82-01A

DATED DECEMBER 1984

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

FOLD ALONG LINE

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

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