



COMPUTER AUTOMATION, INC .

ALPHA-16

BASIC REFERENCE MANUAL

Revised 15 May 1973

Copyright 1973, Computer Automation, Inc.



TABLE OF CONTENTS

Section		Page
Section 1 GENERAL INFORMATION		
1.1	INTRODUCTION.....	1-1
1.1.1	Standard Features.....	1-1
1.1.2	Extended Features.....	1-2
1.1.3	Special Features.....	1-2
Section 2 OPERATING ENVIRONMENT		
2.1	INTRODUCTION.....	2-1
2.2	OPERATING MODES.....	2-1
2.3	LINE EDITING.....	2-2
2.4	PROGRAM EDITING.....	2-2
Section 3 ELEMENTS OF THE LANGUAGE		
3.1	INTRODUCTION.....	3-1
3.2	CONSTANTS.....	3-1
3.3	VARIABLES.....	3-2
3.4	FUNCTION REFERENCE.....	3-2
3.5	OPERATORS.....	3-2
3.6	EXPRESSIONS.....	3-3
3.7	EXPRESSION EVALUATION.....	3-3
Section 4 BASIC STATEMENTS		
4.1	INTRODUCTION.....	4-1
4.2	BASIC-1 STATEMENTS.....	4-1
4.2.1	READ Statement (Executable).....	4-2
4.2.2	DATA Statement (Non-Executable).....	4-3
4.2.3	RESTORE Statement (Executable).....	4-3



TABLE OF CONTENTS (Cont'd)

Section		Page
4.2.4	INPUT Statement (Executable).....	4-4
4.2.5	PRINT Statement (Executable).....	4-4
4.2.6	LET Statement (Executable).....	4-6
4.2.7	GOTO Statement (Executable).....	4-7
4.2.8	GOSUB and RETURN Statements (Executable).....	4-7
4.2.9	FOR and NEXT Statements (Executable).....	4-8
4.2.10	IF Statement (Executable).....	4-9
4.2.11	ON Statement (Executable).....	4-10
4.2.12	STOP Statement (Executable).....	4-10
4.2.13	DIM Statement (Non-Executable).....	4-11
4.2.14	DEF Statement (Non-Executable).....	4-11
4.2.15	REM Statement (Non-Executable).....	4-12
4.2.16	END Statement (Executable).....	4-12
4.3	BASIC-2 and BASIC-3 STATEMENTS.....	4-13
4.3.1	Matrix Statements.....	4-13
4.3.1.1	Matrix Replacement.....	4-13
4.3.1.2	Matrix Addition.....	4-14
4.3.1.3	Matrix Subtraction.....	4-14
4.3.1.4	Matrix Multiplication.....	4-14
4.3.1.5	Scalar Multiplication.....	4-15
4.3.1.6	Matrix Transposition.....	4-15
4.3.1.7	Matrix Inversion.....	4-15
4.3.1.8	Zero Matrix.....	4-15
4.3.1.9	Identity Matrix.....	4-16
4.3.1.10	Constant Matrix.....	4-16
4.3.1.11	Print Matrix.....	4-16
4.3.1.12	Read Matrix.....	4-16
4.3.2	String Operations.....	4-17
4.3.2.1	String Assignment Statement.....	4-18
4.3.2.2	String Relations.....	4-19
4.3.2.3	LEN Function.....	4-19
4.3.2.4	Strings in PRINT Statements.....	4-19
4.3.2.5	Strings in DATA Statements.....	4-19
4.3.2.6	Strings in READ Statements.....	4-19
4.3.2.7	Strings in INPUT Statements.....	4-20
4.3.3	CALL Statement (Executable).....	4-20
Section 5 BASIC COMMANDS		
5.1	INTRODUCTION.....	5-1
5.2	LIST COMMAND.....	5-1



TABLE OF CONTENTS (Cont'd)

Section	Page
5.3 PUNCH COMMAND.....	5-1
5.4 CLEAR COMMAND.....	5-2
5.5 RUN COMMAND.....	5-2
5.6 TAPE COMMAND.....	5-2
5.7 TYPE COMMAND.....	5-3
Section 6 BASIC FUNCTIONS	
6.1 FUNCTIONS.....	6-1
Section 7 ERROR MESSAGES	
7.1 INTRODUCTION.....	7-1
7.2 ERROR MESSAGE FORMAT.....	7-1
Section 8 OPERATING PROCEDURES	
8.1 INTRODUCTION.....	8-1
8.1.1 BASIC-1 Operating Procedure.....	8-1
8.1.2 BASIC-2 Operating Procedure.....	8-1
8.1.2.1 BASIC-2 Absolute Binary Tape (96501-40).....	8-1
8.1.2.2 BASIC-2 Object Tapes.....	8-2
8.1.3 BASIC-3 Operating Procedure.....	8-3
8.1.3.1 BASIC-3 Absolute Binary Tape (96502-40).....	8-3
8.1.3.2 BASIC-3 Object Tapes.....	8-3
8.1.3.3 Table of User Numbers, Device Addresses and Interrupt Locations Used in BASIC-3.....	8-5
8.1.3.4 Limitations on User Machine Language Subroutines Using BASIC-3.....	8-5
Appendix A MACHINE LANGUAGE SUBROUTINES	
A.1 INTRODUCTION.....	A-1
A.2 CODING THE MACHINE LANGUAGE SUBROUTINE.....	A-1



1.1.2 Extended Features

Extended features only available in the BASIC-2 and BASIC-3 Systems include:

1. **String Operations** - This feature enables a user to perform operations on ASCII character strings in many of the ways that numeric constants can be manipulated. For example, one string can be substituted for another, two strings can be concatenated, or strings can be compared.
2. **Matrix Operations** - This feature permits a user to perform mathematical operations on vectors and matrices.
3. **The CALL Statement** - This feature enables a user to call and pass parameters to ALPHA-16 assembly language programs.
4. **(BASIC-3 only) Multiple User Capability** - This feature permits up to nine users, on individual terminals, to run BASIC programs simultaneously.

1.1.3 Special Features

Special features are incorporated in all versions of the ALPHA-16 BASIC System for user convenience, which are not commonly found in other BASIC System implementations. Among these are:

1. **n dimensional arrays** - The ability to define arrays with more than two dimensions.
2. **Multiple Assignments** - The ability to assign a value to two or more variables within a single LET statement.
3. **DIM and DEF Statement Relaxation** - DIM or DEF statements may appear anywhere within a program, and need not precede an array or function reference.



Section 2

OPERATING ENVIRONMENT

2.1 INTRODUCTION

The ALPHA-16 BASIC System as presented here is an on-line program for use from an ASR-33 Teletype Keyboard. (BASIC-3 will also run using a CRT as an I/O terminal). It can operate in either a 4K or 8K single-user configuration. BASIC-1 will operate in a 4K environment; BASIC-2 and BASIC-3 require 8K of memory.

The remainder of this section describes the operating modes of the system, and the user-terminal-system interface.

2.2 OPERATING MODES

After the BASIC System has been loaded into the ALPHA-16's memory, it enters the "command mode". In this mode the system will execute user functions immediately on entry. In the "command mode" the user may:

1. Perform program editing functions as described in Section 2.4,
2. Introduce BASIC commands (see Section 5), or
3. Request the immediate execution of one of the following BASIC statements:

PRINT
LET
GOTO

The above statements are distinguished by lack of their line number. Without a line number, the statement is executed when the RETURN key is typed at the end of a line. The result of the computation is stored for subsequent use, but the statement by which the result was obtained is discarded. This method of operation is commonly referred to as a "desk calculator" mode.

In the "command mode" the system will always prompt the user by typing an asterisk (*) character. When the prompt character appears on a user terminal, this signals that the system is ready to accept one line of input from that terminal.

The system will remain in the "command mode" until a user explicitly directs the system to enter the "execution mode". This is accomplished by typing either a GOTO statement for immediate execution or a RUN command.



The "execution mode" is the mode in which previously entered programs are executed. Once the system enters the "execution mode" it will remain in this mode until one of the following events occurs:

A STOP or END statement is executed

An error is detected

A user initiates a program interrupt by use of:

1. The Console Interrupt (AUTO) switch (BASIC-1 and BASIC-2)
2. The teletype or CRT "Break" key (BASIC-3)

Whenever the system detects any of the above conditions it will return to the "command mode".

2.3 LINE EDITING

User input to the system is on the basis of one line at a time, where a line may consist of a maximum of 72 characters and is terminated by a RETURN character. Since a CRT or TTY keyboard is generally used for input, typing errors may occur. If the user would like the system to ignore all of what he has typed on the current line so that he can start the line afresh, he can terminate the line with a line-feed character rather than the usual carriage-return.

If he would merely like to retype the immediately preceding n characters, leaving the beginning of the line intact, he can type n left arrow (←) characters followed by the corrected characters. For example, if the following sequence of characters is typed

```
*100 Δ Δ PRUN Δ X←←←←INT Δ X,Y,A ← ← ) ← (1)
```

it is equivalent to:

```
*100 Δ Δ PRINT Δ X,Y(1)
```

where Δ designates a blank character.

2.4 PROGRAM EDITING

The BASIC System manages a work area in the computer memory in which the user's BASIC program is maintained. Program editing directives are used for entering a BASIC program into the work area and modifying a program that has previously been entered.



A program editing directive is one of the following formats: a line number (one to four digits) followed by a BASIC statement, a line number only, or two line numbers separated by a comma. The first format is a request to insert a new statement or to replace an existing statement in the user's program. Replacement is done if the supplied line number matches one already existing in the user's program; otherwise, the new statement is inserted in ascending statement number order in the user's program. It is in this way that the user can build up a new program in his work area by repeatedly entering program editing directives with different line numbers. A program editing directive of the second or third format is a request to delete a single existing statement (second format) or a range of existing statements (third format) from the current program.

Thus, at any time when the System has prompted the user with an asterisk (*), the user has the ability to add to, delete from, or otherwise modify his program as it currently stands in his work area by entering a program editing directive. Such modifications are always made on a line-by-line basis.

```
*500  REM  LINE 1
*400  REM  LINE 2
*450  REM  LINE 3
```

the contents of the work area after entry of the third line would be:

```
400  REM  LINE 2
450  REM  LINE 3
500  REM  LINE 1
```

If the user made the entry

```
*500  REM  NEW LINE 1
```

the contents of the work area would be:

```
400  REM  LINE 2
450  REM  LINE 3
500  REM  NEW LINE 1
```



Section 3

ELEMENTS OF THE LANGUAGE

3.1 INTRODUCTION

A BASIC program consists of one or more BASIC lines. A BASIC line consists of a line number followed by a BASIC statement, and terminated by a non-printing carriage return character. A line number consists of from one to four decimal digits in the range from 1 to 9999, and is used to establish the order of the lines in a program. BASIC statements are executed in the ascending order of their line numbers beginning with the lowest numbered executable statement and proceeding through successively higher numbered executable statements except where specifically directed otherwise by control statements such as GOTO, IF, or GOSUB. When writing a BASIC program, it is advisable to number BASIC lines by fives or tens to allow for the possible later insertion of new lines.

A BASIC statement is made up of a sequence of keywords, operands, and operators.

A keyword is a sequence of letters having special significance to the system. In the statement definitions in Section 4, they are shown in capital letters. Examples of keywords are PRINT, INPUT, and LET.

An operand is a variable, a constant, or a function reference. During execution of a BASIC program an operand has a value which is a positive or negative real number whose magnitude is either zero or in the approximate range 2.71×10^{-20} to 9.23×10^{18} . This provides more than six decimal digits of accuracy.

Space characters may be freely used anywhere in a BASIC line to improve the appearance and readability of the BASIC program.

3.2 CONSTANTS

A constant is an item whose value is always defined during execution of a program. Constants may be expressed as integers, decimal numbers, or in exponential format, i.e., a decimal number times some power of ten. A constant consists of a sequence of digits possibly containing a decimal point (.), and possibly followed by a decimal exponent consisting of the letter E, an optional sign, and one or two digits, in that order. The following are examples of constants:

7	Integer
325	Integer
2.6	Decimal number
3.14	Decimal number
.000314E+4	Decimal number in exponential format
314E-02	Decimal number in exponential format



3.3 VARIABLES

A variable is an item, represented by a symbol, that may be assigned a value. A simple variable is either a letter or a letter followed by a digit. A subscripted variable is a letter followed by one or more expressions enclosed in parentheses; if multiple expressions are used, they are separated by commas. For example:

A	Simple variable
B2	Simple variable
X(3)	Subscripted variable
E(A*3)	Subscripted variable
J(Y2,I+5)	Subscripted variable

3.4 FUNCTION REFERENCE

A function reference consists of a three letter function name followed by a parenthesized list of arguments. If there is more than one argument they are separated by commas. The number of arguments supplied in a function reference must agree with the number of arguments supplied in the function definition. Reference to a function produces a value which in general is dependent on (i.e., a function of) the current values of the arguments. The system supports both user-defined and system-defined functions as described in Section 6.

3.5 OPERATORS

An operator is a symbol used in forming an expression. There are two types of operators as listed below:

ARITHMETIC OPERATORS

Symbol	Example	Meaning
+	A+B	Addition
-	A-B	Subtraction
*	A*B	Multiplication
/	A/B	Division
↑	A↑B	Exponentiation



RELATIONAL OPERATORS

Symbol	Example	Meaning
<	A < B	A less than B
>	A > B	A greater than B
<=	A <= B	A less than or equal to B
>=	A >= B	A greater than or equal to B
=	A = B	A equal to B
<>	A <> B	A not equal to B

Operators are said to operate on operands; more precisely, they operate on the current values of the operands.

3.6 EXPRESSIONS

An expression is a sequence of operands and operators, possibly grouped by parentheses. An operand standing by itself is an expression, and if E and F are expressions and @ is any operator, then E @ F, (E), +E, and -E are also expressions.

3.7 EXPRESSION EVALUATION

Within an unparenthesized expression the order of evaluation is:

- Exponentiation
- Multiplication and division
- Addition and subtraction
- Relational

Within a sequence of consecutive operators of the same type, evaluation is from left to right.

Parentheses may be used to override this basic rule for order of evaluation. Parenthesized portions of expressions are evaluated first. Nested parenthesized groups are evaluated beginning with the inner-most grouping, working outward.

All expression evaluation is done in standard floating-point form as defined above. When a pair of operands is operated on by a relational operator, the result is either one or zero depending on whether the relation is true or false, respectively.



The following table shows examples of expressions and the values that would result from their evaluation. It is assumed that the operands X, Y, and Z have the value 4, 3.14, and -2.7, respectively.

<u>Expression</u>	<u>Value</u>
X	4
Z+7	4.3
-2	-2
SGN (Z)	-1
INT (Y+ABS (Z+1))	4
$X \uparrow 2*Z+Y$	-40.06
$X \uparrow 2*(Z+Y)$	7.04
$X < Y < Z$	0

Note that relational operators are not restricted to use in IF statements. For example, the statement

$$\text{LET } X = 5*(Y > 0) + 4*(Y \leq 0)$$

may be used to substitute for a sequence of several statements.



Section 4

BASIC STATEMENTS

4.1 INTRODUCTION

This section defines the general form and operation of the BASIC statements. Section 4.2 defines those statements included in both the BASIC-1 and BASIC-2 Systems, whereas Section 4.3 defines the statements that are only applicable to BASIC-2 Systems.

In the following presentation of the general form of each BASIC statement, the following conventions apply:

1. Elements in capital letters, such as LET, THEN, and GOSUB, are required and must appear exactly as shown.
2. Elements in lower-case are supplied by the user.

LET variable = expression

3. Square brackets surrounding an element of the language indicate that the element is optional and may be included or omitted at the user's option.

INPUT variable [,variable]

4. A right square bracket followed by an ellipsis (...), indicates that the enclosed element may be omitted or repeated an arbitrary number of times.

DATA constant [, constant] ...

4.2 BASIC-1 STATEMENTS

The BASIC-1 language consists of 18 statements. There are two classes of statements: Executable and non-executable.

Non-executable statements are used to specify data items and functions to be used in a program, or to include comments concerning the purpose or operation of a program. The non-executable statements include:

DATA - used to introduce a constant, or a series of constants, into a program.

DIM - used to reserve storage space for arrays.

DEF - used to specify user-defined functions.



REM - used to include program descriptive remarks.

Executable statements control the input and output of data, the evaluation of mathematical formulas, and the control of the program flow. Executable statements include:

READ - used to select constants from DATA statements and assign their values to the variables supplied in READ statements.

INPUT - used to accept data from the input device.

PRINT - used to output formatted data to the output device.

RESTORE - used to permit DATA statement constants to be re-used.

LET - used to assign a value to a variable.

GOTO - used to branch to a specified statement.

GOSUB - used to call a subroutine.

RETURN - used to return from a subroutine.

FOR - NEXT - used in combination to define a program loop.

IF - THEN - used for conditional branching.

ON - used to conditionally select the next statement for execution based on the evaluation of an expression.

STOP - used to return the system to the command mode.

END - used to signify the end of program execution.

4.2.1 READ Statement (Executable)

The READ statement is used in conjunction with DATA statements to permit the assignment of constants to variables as a program is executed. The general form of the READ statement is:

READ variable [, variable] ...

For each variable in the list the next constant is obtained from the consolidated DATA statement list, it is converted to internal floating-point form, and the resulting value is assigned to the variable.



4.2.2 DATA Statement (Non-Executable)

The DATA statement is used to introduce a constant, or a series of constants, into a program. The general form of the DATA statement is:

DATA signed constant [,signed constant] ...

DATA statements establish a sequence of values to be accessed by the READ statements in the program. There may be any number of DATA statements in a program, each containing any number of constants (within the constraints imposed by the length of a BASIC line). When the last constant in a DATA statement has been accessed by a READ statement, the next constant accessed by the same or a different READ statement will be the first constant in the next DATA statement.

```

10  DATA      1, -2, 1.25
20  READ       A, B, C, D, E
30  DATA      50
40  READ       F, G
50  READ       H
60  DATA      20.0E3, 0.155E-10, 252
70  END

```

In the preceding example, the READ statement at line 20 will assign the values 1, -2, 1.25, 50, and 20,000 to the variables A, B, C, D, and E. The READ statement at line 40 will assign the values 0.155 E-10 and 252 to the variables F and G. The READ statement at line 50 attempts to access a value after the last constant in the last DATA statement has been used. The BASIC System will then return an error message, terminate program execution, and return to the command mode.

4.2.3 RESTORE Statement (Executable)

The RESTORE statement permits the re-use of DATA statement constants. The general form of the RESTORE statement is:

RESTORE

Execution of this statement causes the series of constants defined by the DATA statements of the program to be re-used. That is, the first value obtained by the logically next READ statement will be derived from the first constant in the physically first DATA statement in the program.



Example:

```
10 DATA 0,1
20 READ A,B
30 PRINT A,B
40 RESTORE
50 READ E,F
60 END
```

The variables E and F will be assigned the values 0 and 1, respectively.

4.2.4 INPUT Statement (Executable)

The INPUT statement is used to enter data from the input device while a program is running. The general form of the INPUT statement is:

```
INPUT variable [, variable] ...
```

When an INPUT statement is executed, the system prints a prompt character '?' at the user's terminal to indicate that it is ready to accept a line of data. The user then may type in one or more constants formatted exactly the same as specified for the DATA statement (but without the keyword DATA), followed by a carriage-return character. The system then converts the constants into internal floating-point form and assigns the resulting values in order to the variables listed in the INPUT statement. If the user supplies fewer constants than there are variables, the system prints another prompt character and waits for another line of data. If the user supplies more constants than there are variables in the INPUT statement, the extra constants are ignored.

Example:

```
INPUT X1, X2, L(1), L(2), L(3)
```

4.2.5 PRINT Statement (Executable)

The PRINT statement is used to output formatted data on the output device. The general form of the PRINT statement is:

```
PRINT [print element] ...
```

If there is no list following the keyword PRINT, a line is skipped on the user's terminal. If there is a list, the items in it are formatted as described below and printed on one or more lines on the user's terminal.



Each print item consists of a print element followed by a print separator.

A print element is either an expression, a text message, or a TAB function. A text message is a series of characters enclosed in quotation marks. (If it is desired to use the quotation mark as one of the characters, a consecutive pair of quotation marks should be used so as to distinguish it from the terminating quotation mark. The system will recognize a pair of quotation marks and interpret it internally as a single quotation mark.) A TAB function is of the form

TAB(expression)

It may only be used as a print item in a PRINT statement.

A print separator is either a comma (,) or a semicolon (;).

Each expression or text message in the list is translated into a field in a print line. The first field of the first PRINT statement starts at column position 0, (print positions are numbered from 0 through 71). The beginning position of subsequent fields depends on the previous print separator, which may have been in the previous PRINT statement. If the separator was a comma, the field begins in the next available zone. Zones begin at column positions 0, 15, 30, 45, and 60. If the separator was a semicolon, the field begins at the next position following the end of the previous field. However, if the print item is a TAB function, it overrides the effect of the following separator and the truncated value of the expression becomes the beginning column position of the next field. If the TAB value is greater than 71, the next field begins at column position zero of the next line.

If the starting column position of a field is such that it cannot be completely contained on the current line, it is printed beginning in column zero of the next line.

The translation of a text message into a field in the print line is straightforward; the character string is used verbatim except for the replacement of double quotation marks by a single quotation mark.

The translation of an expression into a field in the print line is more complex. The expression is evaluated. Then, if the resulting value is an integer of a magnitude less than 10^6 , the following characters are placed in the field from left to right:

- a minus sign (-) if the value is negative, or a space otherwise.
- one to six decimal digits representing the value of the expression.
- two spaces



If the value is not an integer but its magnitude is greater than or equal to 0.1 and less than 10^6 , its output format is the same as for an integer but with a decimal point inserted at the appropriate point within the string of decimal digits.

For other values the following sequence of characters is placed in the print line from left to right:

- a minus sign if the value is negative, or a space otherwise.
- the most significant decimal digit of the value.
- a decimal point(.).
- the next five decimal digits of the decimally scaled value of the expression.
- the letter E.
- a plus (+) or minus (-) sign depending on the sign of the decimal exponent.
- two decimal digits representing the value of the decimal exponent.
- two spaces.

Examples:

```
10 LET X = 10.52
15 PRINT "X = "; X; TAB (14), X*1000, -X/1000
```

The preceding statements would cause the following printing to be done:

```
XΔ = ΔΔ 10.52 ΔΔΔΔΔ 10520ΔΔ ΔΔ ΔΔ ΔΔ ΔΔ ΔΔ -1.052E-02
```

The deltas (Δ) represent space characters.

4.2.6 LET Statement (Executable)

The LET statement is used to assign a value to a variable. The general form of the LET statement is:

```
LET variable = [variable=] ...expression
```



The expression on the right is evaluated and the resulting value becomes the current value for each of the variables appearing to the left of an equals sign.

Examples:

```
10 LET X=4+Z1
20 LET A(1)=A(2)=0
30 LET W=X=Y+Z
40 LET W=X+Y=Z
```

Note that in line 30 the second equals sign acts as a replacement (assignment) operator, whereas in line 40 the second equals sign acts as a relational operator.

Expressions appearing as subscripts are evaluated before any value replacements are done. For example, if the current value of J is 4, the assignment statement

```
LET X(J-1)=J=3
```

will replace the value of X(3), not X(2).

4.2.7 GOTO Statement (Executable)

The GOTO statement is used to unconditionally alter the sequence of statement execution. The general form of the GOTO statement is:

```
GOTO line number
```

When control reaches a GOTO statement during the execution of the user's program, control is unconditionally transferred to the statement having that line number.

Example:

```
10 INPUT X
20 PRINT EXP (X)
30 GOTO 10
40 END
```

Execution of this program converts the user's terminal into a device which automatically types out the value of e^x each time a value (X) is typed in.

4.2.8 GOSUB and RETURN Statements (Executable)

The GOSUB and RETURN statements give the BASIC language a subroutine capability. The general form of the GOSUB and RETURN statements is:

```
GOSUB line number
```

```
RETURN
```



Execution of a GOSUB statement transfers control to the specified line number. Execution of a RETURN statement sends control to the first executable statement following the most recently executed GOSUB statement for which a RETURN has not yet been executed.

Example:

```
10 LET X=5
20 GOSUB 60
30 LET X=7
40 GOSUB 60
50 STOP
60 LET A(I) =X
70 LET I=I+1
80 RETURN
```

Subroutines may also be nested; i.e., subroutines may contain calls to other subroutines. Such nesting can be carried to any level. The following example illustrates subroutine nesting with a shared RETURN statement.

```
20 GOSUB 30
25 STOP
30 LET X=5
31 LET Y=7
32 GOSUB 40
33 LET X=X 2
34 LET Y=Y 2
35 GOSUB 40
36 LET X=X*5
37 LET Y=Y*9
40 PRINT X,Y
45 RETURN
```

4.2.9 FOR and NEXT Statements (Executable)

The FOR and NEXT statements are used to specify the beginning and ending points of program loops. The general form of the FOR and NEXT statements is:

```
FOR simple variable=expression TO expression STEP expression
NEXT simple variable
```

The FOR statement in conjunction with the NEXT statement provides the user with a convenient mechanism for defining program loops in which a simple variable varies linearly as the loop proceeds. The association between FOR and NEXT statements is established by the identifier of the simple variable. FOR and NEXT statements must



be paired within a program, and the set of statements through which control passes between them is called the range of the FOR statement. Referring to the first, second, and third expressions in the FOR statement as X, Y, and Z, respectively, X is the first initial value of the simple variable for the first execution of the range. Z specifies the value to be added to the current value of the simple variable each time control reaches the associated NEXT statement. If Z is missing from the FOR statement a value of +1 is used. If Z is specified, it must be preceded by the keyword STEP. Then if the quantity $(Y-X)/Z$ is not less than zero, control is passed to the first executable statement following the FOR statement; otherwise, control passes to the first executable statement following the NEXT statement. The values used are the ones obtained when the FOR statement is first encountered. The range of the FOR statement is always executed at least once even if the quantity $(Y-X)/Z$ is less than zero to start with.

The following is an example of nested FOR statements:

```

10  DIM           A(20,20), B(20,10), C(10,20)
20  FOR I =       1 to 20
30  FOR J =       1 to 20
40  LET          A(I,J)=0
50  FOR K =       1 to 10
60  LET          A(I,J)=A(I,J)+B(I,K)*C(K,J)
70  NEXT K
80  NEXT J
90  NEXT I
100 END

```

4.2.10 IF Statement (Executable)

The IF statement is used to provide a conditional program branch. The general form of the IF statement is:

IF expression THEN line number

When an IF statement is executed, the expression is evaluated and interpreted as a logical value. If the value is false, control is transferred to the next executable statement. If the value is true, control is transferred to the statement having the specified line number. A value of zero is false; any non-zero value is true.

The expression usually takes the form $E@F$, where @ is one of the relational operators, and E and F are expressions.



```
10 INPUT X
15 IF X <=0 THEN 30
20 PRINT LOG (X)
25 GOTO 10
30 PRINT "POSITIVE VALUES REQUIRED"
35 GOTO 10
40 END
```

4.2.11 ON Statement (Executable)

The ON statement is used as a conditional GOTO statement to alter the sequence of statement execution. The general form of the ON statement is:

ON expression GOTO line number [,line number] ...

When an ON statement is executed, the expression is evaluated and the result is truncated to an integer. The integer is then used as an index to select one of the line numbers in the list. Control is transferred to the statement having the selected line number. If the integer is less than one or greater than the number of line numbers specified, control passes to the statement after the ON statement.

The ON statement gives the user a multi-directional switch capability.

```
2 INPUT X
4 ON SGN (X)+2 GOTO 6, 10, 14
6 PRINT "IMAGINARY"
8 GOTO 2
10 PRINT "UNDEFINED"
12 GOTO 2
14 PRINT LOG (X)
16 GOTO 2
```

4.2.12 STOP Statement (Executable)

The STOP statement is used to terminate program execution. The general form of the STOP statement is:

STOP

Upon encountering a STOP statement during the execution of a program, execution ceases and the system reverts to the command mode of operation. (See Section 2).



STOP statements can be used anywhere in a program to designate the logical rather than the physical end of a program. STOP is useful in debugging programs, and in suspending program execution at critical points where a user may wish to display intermediate results, and then alter program execution depending on the displayed results.

4.2.13 DIM Statement (Non-Executable)

The DIM statement is used to reserve storage space for data arrays. The general form of the DIM statement is:

```
DIM letter(constant [,constant] ...) ,letter(constant [,constant] ...) ...
```

The DIM statement defines the name and dimensionality of arrays. Each "letter" appearing in a DIM statement becomes the name of an array, so that a maximum of 26 arrays may be declared in a program. The dimensionality of an array is determined by the number of constants in parentheses following the name. Each constant must be a positive integer, and the value of the constant defines the largest value that a subscript expression in that position may have. The minimum value that any subscript expression may have is zero. Every reference to an element of the array must contain exactly as many subscript expressions as there are constants in the declaration of the array. For example, the statement

```
DIM P(5), Q(8,6,4), R(2,2,2,2,3)
```

defines three arrays, containing 6, 315, and 324 elements, respectively. Q is a 9x7x5 array, and references elsewhere in the program to elements of Q must carry a parenthesized list of three subscript expressions. The reference Q(0,0,0) refers to the first element of Q. P is a one-dimensional array (vector).

It is permissible for an array name to be the same as the name of a simple variable.

DIM statements may appear anywhere in the program, and they are not required to precede references to the arrays they define.

4.2.14 DEF Statement (Non-Executable)

The DEF statement enables the introduction of a user defined function. The general form of the DEF statement is:

```
DEF FN letter(simple variable [,simple]variable ...) = expression
```



The DEF statement gives the user a convenient means of avoiding rewriting expressions which occur frequently in his program and differ only in their operands. The "FN letter" combination becomes the name of an explicitly defined function which may then be used whenever a function reference is permitted. Such a reference causes the evaluation of the names shown in the definition. The 'letter' in the function name may be any of the 26 alphabetic characters, thus giving the user a potential repertoire of 26 explicitly defined functions.

The DEF statement which defines a function name need not precede the reference.

It is permissible (although not necessarily desirable) for the argument names in the definition to be the same as names used elsewhere in the program.

Example:

Using the recursive definition

$$X_{m+1} = \frac{1}{2}(X_m + Y/X_m)$$

make three iterations starting with $X_1 = 1$.

```

10 DEF FNQ(C)=(C+Y/C)/2
20 LET Y=4
30 LET X=FNQ(FNQ(FNQ(1)))

```

4.2.15 REM Statement (Non-Executable)

The REM is used to insert notes and comments in user programs. The general form of the REM statement is:

REM arbitrary character string

REM statements have no effect on the execution of a program, however, they do take up storage space.

4.2.16 END Statement (Executable)

The END statement is used to designate the physical end of a program. The general form of the END statement is:

END



If control reaches an END statement during the course of executing a program, execution ceases and the System reverts to the command mode of operation. (See Section 1.) For purposes of compatibility with other BASIC Systems, every program should have an END statement and it should be the last statement in the program. However, neither of these requirements is present in the ALPHA-16 BASIC System.

An END statement is operationally equivalent to a STOP statement.

4.3 BASIC-2 and BASIC-3 STATEMENTS

The BASIC-2 and BASIC-3 Systems include all of the BASIC-1 statements defined in Section 4.1, plus the statements defined below. This includes twelve matrix statements, and provisions for accepting string operands in the LET, PRINT, DATA, READ, and INPUT statements.

4.3.1 Matrix Statements

A matrix is a two-dimensional array. That is, a letter that appears in a DIM statement with two constants in the following parentheses. Twelve different matrix statements are available in the BASIC-2 and BASIC-3 Systems. Each of these statements performs a complete matrix operation, thus relieving the user of the necessity of writing the usual FOR/NEXT loops that would otherwise be required.

None of the matrix operations refer to row or column zero of the matrix operands. They all operate on the matrices beginning at row and column one.

The general form and the effect of executing each of the twelve matrix statements is defined below. The reader is presumed to have a knowledge of matrices and matrix operations.

4.3.1.1 Matrix Replacement

General Form:

`MAT matrix=matrix`

Effect:

Each element of the first matrix is assigned the current value of the corresponding element of the second matrix. The two matrices must have the same dimensions.



4.3.1.2 Matrix Addition

General Form:

$$\text{MAT matrix}=\text{matrix}+\text{matrix}$$

Effect:

Each element of the first matrix is assigned a value which is the sum of the corresponding elements of the other two matrices. All three matrices must have the same dimensions.

4.3.1.3 Matrix Subtraction

General Form:

$$\text{MAT matrix}=\text{matrix}-\text{matrix}$$

Effect:

Each element of the first matrix is assigned a value which is the difference between the corresponding elements of the second and third matrices. All three matrices must have the same dimensions.

The preceding three kinds of matrix statements (replacement, addition, and subtraction), may be used with matrices of any dimension as long as they match.

4.3.1.4 Matrix Multiplication

General Form:

$$\text{MAT matrix}=\text{matrix}*\text{matrix}$$

Effect:

The elements of the first matrix are assigned values such that it is the product of the other two matrices according to the usual rule for matrix multiplication. The standard restrictions concerning the dimensions of the three matrices apply. That is, if x , y , and z represent integer constants, then the following relationship must exist between the dimensions of three matrices A , B , and C :

$$\begin{aligned} &\text{DIM } A(x, z), B(x, y), C(y, z) \\ &\text{MAT } A=B*C \end{aligned}$$

The matrix on the left side of the equals sign must not be the same as either of the matrices on the right.



4.3.1.5 Scalar Multiplication

General Form:

$$\text{MAT matrix} = (\text{expression}) * \text{matrix}$$

Effect:

Each element of the first matrix is assigned a value which is the product of the value of the expression and the corresponding element of the second matrix. The two matrices must have the same dimensions.

4.3.1.6 Matrix Transposition

General Form:

$$\text{MAT matrix} = \text{TRN}(\text{matrix})$$

Effect:

The first matrix is made the transpose of the second matrix. The two matrices must be square and have the same dimensions; they may be the same matrix.

4.3.1.7 Matrix Inversion

General Form:

$$\text{MAT matrix} = \text{INV}(\text{matrix})$$

Effect:

The elements of the first matrix are assigned values such that it is the inverse (in the usual mathematical sense) of the second matrix. The two matrices must be square and have the same dimensions.

4.3.1.8 Zero Matrix

General Form:

$$\text{MAT matrix} = \text{ZER}$$

Effect:

The value of each element of the matrix is set to zero.



4.3.1.9 Identity Matrix

General Form:

MAT matrix = IDN

Effect:

The value of every element of the matrix is set to zero except for the elements on the principle diagonal which are set to one. The matrix must be square.

4.3.1.10 Constant Matrix

General Form:

MAT matrix=CON

Effect:

The value of each element of the matrix is set to one.

4.3.1.11 Print Matrix

General Form:

MAT PRINT matrix [separator matrix] ... separator

Effect:

The value of every element of each of the elements in the list is printed in the format described under the PRINT statement. Each separator is either a comma or a semicolon, and it governs the spacing on the print line between each matrix element in the same way as for ordinary printing. The matrix elements are ordered row-wise, and each row is started in a new line.

4.3.1.12 Read Matrix

General Form:

MAT READ matrix [,matrix] ...



Effect:

Every element of each matrix in the list is assigned a value taken from a constant in the consolidated DATA statement list. The matrix elements are ordered row-wise.

4.3.2 String Operations

A string is a sequence of characters. The number of characters in the sequence is the length of the string. The length of a string can vary dynamically during the execution of a program as long as it does not exceed the maximum allowable value. The maximum length of a string is specified in a DIM statement by a single constant in parentheses after the string name. The length of a string may not exceed 72.

A string operand is either a string constant, a string variable, or a partial string. A string constant is a sequence of arbitrary characters preceded and followed by quotation marks. (If it is desired to include the quotation mark character in the string, it can be indicated by writing a pair of consecutive quotation marks.) The length of a string constant is the number of characters inside the delimiting quotation marks, interior double quotation marks counting as a single character. A string variable is a letter followed by a dollar sign (\$). A partial string is a string variable followed by one or two expressions enclosed in parentheses. (If two expressions are used they are separated by a comma.) A partial string selects a consecutive sequence of characters from the string variable as a substring. The truncated value of the first expression specifies the character position of the last character of the substring. If the second expression is missing, the substring extends to the current end of the string variable. The first character of a string is referenced by a subscript value of zero.

Examples:

"XYZ" "456"	A seven-character string constant.
X\$	A string variable.
P\$(4,8)	A partial string consisting of character 4, 5, 6, 7, and 8 from the string variable P\$.
P\$(65)	A partial string taken from the end of the string variable P\$.

If n is the current working length of the string variable W\$, then a reference to W\$ is the same as a reference to W\$(1,n).



A string expression is either a single string operand or a sequence of two or more string operands separated by the concatenation operator (+). Two string operands separated by a concatenation operator define a new string whose length is the sum of the lengths of the two constituent strings. The character of which the new string is composed are the characters from the string on the left side of the concatenation operator followed by the characters from the string on the right side of the operator.

A string relation is a pair of string expressions separated by any one of the relational operators defined in Section 3. A string relation has a value one or zero (true or false) depending on whether the relation is true or false. Strings are compared one at a time beginning at the left end of each string and continuing rightward until an unequal pair of characters is found. (If the end of one string is reached before the end of the other, a space character is supplied to form the pair instead of a character from the shorter string.) If the end of the longer string is reached and no unequal character pairs are found, the strings are considered to be "equal". If an unequal pair of characters is found, the characters are ranked according to the collating sequence shown below, and the string contributing the "higher" character is considered to be "greater" than the other string. The collating sequence, from low values to high values, is:

;/;"#\$\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ\ [/] ↑ ←

The BASIC-2 and BASIC-3 Systems provide the user with the following facilities:

4.3.2.1 String Assignment Statement

General Form:

LET string variable =string expression

Effect:

The string expression on the right is constructed and the resulting character string is copied into the string variable beginning at the first character position. The working length of the string variable on the left is set equal to the length of the string expression unless this would increase it beyond its maximum. In this case, the working length is set to the maximum and surplus characters are truncated from the right end of the string expression.

The operand on the left of the equals sign may also be a partial string. Characters are moved into the partial string beginning at the character position specified by the first expression and continuing rightward to the character position specified by the second expression or the last character of the string expression,



whichever comes first. Space characters are supplied on the right if the length of the string expression is less than the length of the partial string. If the length of second expression in the partial string is greater than the current working length of the string, the working length is set equal to the value of the second expression; otherwise, the working length is left unchanged.

4.3.2.2 String Relations

A string relation, as defined above, may be used as an operand in an expression in the same way as an arithmetic relation.

4.3.2.3 LEN Function

An additional function is available. Its name is LEN, its argument is a string expression, and its value is the current working length of the string expression. The LEN function may be used just as any other function reference.

4.3.2.4 Strings in PRINT Statements

A string operand may be used as an element in a print list. The current working length of the string operand determines the number of characters placed on the print line.

4.3.2.5 Strings in DATA Statements

A string constant may appear in the list of a DATA statement. The logic of the program must be such as to pair the string constant with a string variable in a READ statement, since it is not permitted to read a string constant into a numeric variable.

4.3.2.6 Strings in READ Statements

A string variable or a partial string may appear as an element in a READ list. The logic of the program must be such as to pair the string variable (or partial string) with a string constant in a DATA statement, since it is not permitted to read a numeric value (constant) into a string. The effect of executing a READ statement containing a string operand is the same as executing a string assignment statement with the string operand on the left and the string constant on the right of the equals sign.



4.3.2.7 Strings in INPUT Statements

String variables and partial strings may be used in an INPUT list just as in a READ statement, with the same effect. They must be paired with string constants in the input line typed in by the user.

4.3.3 CALL Statement (Executable)

The CALL statement is available to call and pass parameters to and from ALPHA-16 assembly language subroutines. The general form of the CALL statement is:

```
CALL (n [expression] ...)
```

where n is an expression whose truncated value identifies a specific user written machine language program previously linked to the BASIC-2 processor. When this statement is executed, the expressions (if any) are evaluated and the resulting values are supplied to the machine language program as arguments.

Refer to Appendix A - MACHINE LANGUAGE SUBROUTINES for a detailed description of their implementation and usage.



Section 5

BASIC COMMANDS

5.1 INTRODUCTION

The BASIC System accepts six commands for controlling a user's operating environment. Each command consists of a unique keyword of from three to five letters. Commands are not preceded by a line number, and are always executed immediately upon entry. Square brackets surrounding an element of the line indicate that parameter is optional and may be included or omitted as required. The six commands are:

- LIST - List all or part of a user's program
- PUNCH - Output all or part of a user's program to paper tape
- CLEAR - Initialize a user's word area
- RUN - Begin execution of a user's program
- TAPE - Accept program input from paper tape
- TYPE - Accept program input from the keyboard

5.2 LIST COMMAND

The LIST command is used to print all or part of a user's BASIC program as it currently stands in a user's work area. The general form of the LIST command is:

LIST [line number [, line number]]

If there are no line number parameters following the keyword, the entire program is printed. If only a single line number is given, that line is printed. If the command contains two line number parameters, that portion of the program whose line numbers lie between the values is printed.

Although the user may have originally entered his program statements in any order, they are maintained in the work area in order of ascending line number, and this is the order in which they are listed.

5.3 PUNCH COMMAND

The PUNCH command is used to output all or part of a user's BASIC program as it currently stands in the user's work area to paper tape. The general form of the PUNCH command is:

PUNCH [line number [,line number]]



Setting DATA SWITCH 0 will cause punching on the high speed punch; resetting the DATA SWITCH will cause punching on the teletype paper tape punch. (Note: This applies to BASIC-1 and BASIC-2 only. BASIC-3 outputs to the teletype punch only.)

The format of the program on paper tape is such that it can be read back into the user's work area at a later time for execution or further modification.

The rules for PUNCHing out part of a program are the same as for LISTing a user's program.

5.4 CLEAR COMMAND

The CLEAR command is used to initialize a user's work area. The general form of the CLEAR command is:

CLEAR

When a CLEAR command is entered a user's work area is cleared of all program text, and his relation with the BASIC System is re-initialized. After execution of a CLEAR command the only meaningful action on the part of the user is a series of program editing directives to construct a new program in his work area.

5.5 RUN COMMAND

The RUN command is used to begin execution of a user's BASIC program. The general form of the RUN command is:

RUN

Upon receiving a RUN command the BASIC System leaves the command mode of operation and enters the execution mode. The user's program as it currently stands in the work area is executed statement by statement, beginning with the first (lowest numbered) executable statement, and continuing until a STOP or END statement is encountered, or until an abnormal condition occurs. At this point a message identifying the cause and location of the termination is printed, the System re-enters the command mode of operation and prints the asterisk prompt character.

5.6 TAPE COMMAND

The TAPE command is used to inform the BASIC System that a BASIC program is to be input from paper tape. The general form of the TAPE command is:

TAPE



Upon receiving this command the BASIC System will prepare to accept all subsequent program input from a paper tape reader. Setting Data Switch 0 will cause reading from the high speed reader; resetting the data switch will cause reading from the teletype paper tape reader. (Note: This applies to BASIC-1 and BASIC-2 only. BASIC-3 reads from the teletype tape reader only).

5.7 TYPE COMMAND

The TYPE command is used to inform the BASIC System that a BASIC program is to be input from a terminal keyboard. The general form of the TYPE command is:

TYPE

The BASIC System normally expects to receive program input from a terminal keyboard, and so the TYPE command is only required during a user session following the use of a TAPE command which has re-assigned the input source to a paper tape reader.

The keyboard mode of operation may be re-entered by depressing the console interrupt (AUTO) Switch (BASIC-1 and BASIC-2), or the Teletype or CRT "BREAK" key (BASIC-3).



Section 6

BASIC FUNCTIONS

6.1 FUNCTIONS

The BASIC System includes 13 pre-defined functions to perform mathematical operations. A function reference consists of a three letter function name followed by a parenthesized list of arguments. If there is more than one argument, they are separated by commas. The number of arguments supplied in a function reference must agree with the number of arguments specified in the function definition. Reference to a function produces a value which in general is dependent on (i.e., a function of) the current values of the arguments.

COMMON FUNCTIONS:

EXP	(X)	E raised to the X power
LOG	(X)	Natural logarithm of X
ABS	(X)	Absolute value of X
INT	(X)	The largest integer not greater than X
MOD	(X,Y)	X modulo Y
SGN	(X)	A value indicating the sign of X
SQR	(X)	Square root of X

BASIC-2 and BASIC-3 FUNCTIONS:

SIN	(X)	Sine of X (X is expressed in radians)
COS	(X)	Cosine of X (X is expressed in radians)
TAN	(X)	Tangent of X (X is expressed in radians)
ATN	(X)	Arctangent of X is returned as an angle in radians in the range $\pm \pi/2$.
RND	(X)	A random number between 0 and 1
LEN	(X\$)	Return the length of the string variable X\$

The arguments X and Y to the functions may be a constant, a variable, an expression, or another function.

The value of the argument to the RND function governs the generation of random numbers in the following way: if the value is zero then the random number is generated from the previously generated random number (or from a fixed value if this is the first request for a random number within the current execution);



if the value is non-zero then it takes the place of the previously generated number (or the fixed value) in the algorithm that produces the new random number. In this way, the user has the option of always generating the same sequence of random numbers (for check-out purposes, perhaps) or of generating a different sequence for each execution of his program.

The SGN function returns the value -1, 0, or +1, depending on whether the value of the argument is negative, zero, or positive.

In addition to the 13 pre-defined functions listed above, the ALPHA-16 BASIC System supports user-defined functions, where the function reference is of the form:

FNa	(X)	User-defined function; a is any letter; and the function must be defined in a DEF statement.
-----	-----	--



Section 7

ERROR MESSAGES

7.1 INTRODUCTION

When the BASIC System detects an error while scanning or processing a line of input from the user or while executing a BASIC statement, it prints an error message at the user's terminal (and reverts to the command mode of operation if it was in the execute mode).

7.2 ERROR MESSAGE FORMAT

An error message has the following format:

```
//aa dd
```

The aa field is a two-letter code identifying the error detected (as shown in the list below), and the dd field contains two decimal digits which locate the character position within the line at which the error was detected. The BASIC language statement containing the error is printed on the next line following the error message. The list of possible error codes and their meanings is as follows:

AR	Arrays not conformable. A matrix operation has been specified and the dimensions of the matrix operands are such that the operation is not possible.
CA	No such CALL subroutine.
CN	Constant missing in Data statement or input line.
DA	Doubly-defined array.
DC	List of DATA constants exhausted.
DF	Doubly-defined function. The same function name has appeared in more than one DEF statement.
DZ	Division by zero attempted.
ER	Syntax error. The statement or command does not have the proper structure.
ET	Program has reached end of text.



EX	Extraneous characters on line. The initial characters on the line are interpretable as a legal statement or command, but there are further unusable non-space characters.
FA	Function argument error. The number of arguments in the function reference is not the same as in the function definition.
FO	Improper FOR variable. The operand following FOR must be a simple variable.
HL	Halt. A STOP or END statement has been encountered during execution.
IA	Illegal assignment.
ID	Illegal dimension specifier.
IM	Square matrix required.
IN	Improperly formed constant. Most likely cause of this error is missing digits in the exponent.
IO	Illegal operator with string operand.
IS	Illegal value for a statement number.
LG	Negative or zero argument to LOG function.
LN	Statement number not allowed.
LS	LEN function requires a string argument.
M=	No replacement operator in LET, DEF or FOR statement.
M(Unmatched right parenthesis. There is an excess of left parentheses.
M)	Unmatched left parenthesis. There is an excess of right parentheses.
MA	Unrecognized MAT statement.
MM	Mode mixing between string and numeric data.



MO	No operand. The syntax of the statement requires an operand in this position.
MS	Missing statement. A reference to a non-existent line number has been encountered in a GOTO, GOSUB, IF, or ON statement.
MT	No program text. A LIST or PUNCH command has been given and there are no BASIC statements in the user's work area.
NL	Statement number required on this statement.
NO	Overflow on conversion to integer format.
NS	Deletion of non-existent statement number.
NX	NEXT statement encountered without having executed an associated FOR statement.
OV	Numerical overflow. The value of a constant or an intermediate value of an expression has exceeded the maximum magnitude (approximately 9.23×10^{18}).
RT	RETURN statement encountered while no GOSUB statement is active.
SA	Illegal string argument.
SM	Attempted inversion of a singular or nearly singular matrix.
SN	Wrong number of subscripts in an array reference.
SO	String overflow. The working length of a string has exceeded the maximum length.
SQ	Negative argument to SQR function.
SR	Subscript range error. A subscript has been encountered which refers to an array element outside the array.
SS	Substring subscripts contradictory. The second argument of a substring reference is less than the first argument.
ST	String operation not available.
TD	Two-dimensional array required.



UF	Undefined function. A reference has been made to a function which is not in the library and has not been defined in a DEF statement.
UM	Undefined matrix in MAT statement.
UQ	Unbalanced quotation marks.
US	The input line is not recognizable as either a statement or command.
WA	User's work area has been exhausted. If this error occurs while attempting to add a new line to a program, the only remedy is to reduce the size of the program. If the error occurs during program execution, it may be possible to avoid it by breaking up long expressions into two or more smaller expressions or by reducing the size of arrays.
XI	Excess input characters.
XP	Expression not permitted here.



Section 8

OPERATING PROCEDURES

8.1 INTRODUCTION

The BASIC Processor program must be loaded into the ALPHA-16 computer for execution. This section describes the procedures required to load and execute the BASIC processors.

8.1.1 BASIC-1 Operating Procedure

The BASIC-1 Processor is delivered in absolute binary tape format and can be loaded by BLD, LAMBDA, or AUTOLOAD. BASIC-1 requires a minimum of 4K words of memory.

1. The BASIC-1 system tape (96500-40) is readied in the selected reader and loaded with BLD, LAMBDA (set A = : 100, X = : 0) or AUTOLOAD.
2. Execution begins by entering : 0100 in the P register and depressing RUN. (The STOP and SENSE switches must be up.)
3. BASIC will respond by printing its name on the teletypewriter, followed by the asterisk (*) prompt character. Refer to section 2 (Operating Environment) for command entry procedures.
4. The CLEAR command should be the first command of each BASIC Program to "initialize" BASIC and insure sufficient buffer storage for the user program.

8.1.2 BASIC-2 Operating Procedure

The BASIC-2 System is delivered as both absolute binary (for convenient loading) and relocatable object (for inclusion of machine language subroutines) tapes.

8.1.2.1 BASIC-2 Absolute Binary Tape (96501-40)

1. The BASIC-2 system tape is readied in the selected reader and loaded with BLD, LAMBDA (set A - : 100, X = : 0) or AUTOLOAD.
2. Execution begins by entering : 0100 in the P register and depressing RUN. (The STOP and SENSE switches must be up.)



3. BASIC will respond by printing its name on the teletypewriter, followed by the asterisk (*) prompt character. Refer to section 2 (Operating Environment) for command entry procedures.
4. The CLEAR command should be the first command of each BASIC Program to "initialize" BASIC and insure sufficient buffer storage for the user program.

8.1.2.2 BASIC-2 Object Tapes

The BASIC-2 Processor is delivered as two object format tapes:

1. BASIC-2 Processor (96501-30)
2. IOC/EBASIC Module (96501-31)

and must be loaded with the LAMBDA Object Loader. BASIC-2 requires a minimum of 8K words of memory, and will use any additional memory available.

The first program loaded is the BASIC-2 Processor (96501-30), then the user's machine language subroutines, if any, and finally, the IOC/EBASIC library tape (96501-31). A machine language entry which is not resolved by loading a user subroutine will be listed as undefined ('U' error), when the library tape is finished loading. This can be ignored, the BASIC Processor will generate a "CA" error if these routines are referenced.

1. The BASIC-2 processor tape (96501-30) is readied in the selected reader and the object program is loaded by LAMBDA (set A - :100, X - :65).
2. Any user written machine language subroutines may now be loaded.
3. Finally, the library mode BASIC-2 IOC/EBASIC tape (96501-31) is loaded. There should now be no undefined references, except for unused ML machine language routines.
4. Execution begins by entering :0100 in the P register and depressing RUN. (The STOP and SENSE switches must be up.)
5. BASIC-2 will respond by printing its name on the teletypewriter, followed by the asterisk (*) prompt character. Refer to section 2 (Operating Environment) for command entry procedures.
6. The CLEAR command should be the first command of each BASIC program to "initialize" BASIC and insure sufficient buffer storage for the user program.



8.1.3 BASIC-3 Operating Procedure

The BASIC-3 System is delivered in both absolute binary (for convenient loading) or object (for inclusion of machine language subroutines) tapes.

8.1.3.1 BASIC-3 Absolute Binary Tape (96502-40)

1. The BASIC-3 system tape is readied in the selected reader and loaded with BLD, LAMBDA (set A - : 100, X = : 0) or AUTOLOAD.
2. Execution begins by entering : 0100 in the P register and depressing RUN. (The STOP switch must be up.)
3. BASIC-3 will respond by printing its name on the console teletype or CRT, followed by the query "INPUT USERS:". The user should respond by entering the number, from 1 to 9, of each user who will access the BASIC-3 program. (Refer to section 8.1.3.3 for the table of user numbers and their associated device addresses and interrupt locations.) User numbers must be separated by commas, and the response must be terminated with a carriage return. Input of a carriage return only signifies that all nine users are to be on-line. Input of an illegal character or separator will cause the query to be repeated.
4. Upon completion of user number input, the title "ALPHA-16 BASIC-3" is output to each on-line terminal, followed by the asterisk (*) prompt character. Refer to section 2 (Operating Environment) for command entry procedures.
5. The CLEAR command should be the first command of each BASIC program to initialize BASIC and insure sufficient buffer storage for the user program.

8.1.3.2 BASIC-3 Object Tapes

The BASIC-3 system is delivered as four object format programs and must be loaded with the LAMBDA Object Loader. BASIC-3 requires a minimum of 8K words of memory, and will use any additional memory available.

The four delivered programs consist of:

1. MONITOR/IOC Module (96502-31)
2. TUP (96014-30)
3. BASIC-3 Processor (96502-30)
4. EBASIC Module (96500-32)



and should be loaded with LAMBDA in the order shown above. User machine-language subroutines, if any, may be loaded any time after MONITOR/IOC and before EBASIC.

NOTE

See section 8.1.3.4 for limitations on user machine-language subroutines.

A machine language reference not resolved by loading a user subroutine will be listed as undefined ('U' error) when the program tapes finish loading. This can be ignored, and the BASIC processor will generate a "CA" error if these routines are referenced.

1. The MONITOR/IOC tape (96502-31) is readied in the selected reader and the object program is loaded by LAMBDA (set A = : 100, X = : 63).
2. Any user written machine language subroutines may be loaded now, or after steps 3 or 4 (see section 8.1.3.4 for limitations on machine-language subroutines).
3. The Teletype Utility Package (TUP) tape (96014-30) is loaded.
4. The BASIC-3 processor tape (96502-30) is loaded.
5. The EBASIC tape (96500-32) is loaded last. There should now be no undefined references, except for unused ML machine language routines.
6. Execution begins by entering : 0100 in the P register and depressing RUN. (The STOP switch must be up.)
7. BASIC will respond by printing its name on the Teletype or CRT, followed by the query "INPUT USERS:". The user should respond by entering the number, from 1 to 9, of each user who will access the BASIC-3 program. (Refer to section 8.1.3.3 for the table of user numbers and their associated device addresses and interrupt locations.) User numbers must be separated by commas, and the response must be terminated with a carriage return. Input of a carriage return only signifies that all nine users are to be on-line. Input of an illegal character or separator will cause the query to be repeated.
8. Upon completion of user number input, the title "ALPHA-16 BASIC-3" is output to each on-line terminal, followed by the asterisk (*) prompt character. Refer to section 2 (Operating Environment) for command entry procedures.
9. The CLEAR command should be the first command of each BASIC program to initialize BASIC and insure sufficient buffer storage for the user program.



8.1.3.3 Table of User Numbers, Device Addresses and Interrupt Locations Used in BASIC-3

<u>User No.</u>	<u>Device Address</u>	<u>Interrupt Location</u>	<u>Dual TTY Controller</u>	<u>Unit within Dual TTY Controller</u>
1	:07	:02	(standard console teletype/CRT)	
2	:03	:22	1	0
3	:02	:2A	1	1
4	:0A	:32	2	0
5	:0B	:3A	2	1
6	:0C	:42	3	0
7	:0D	:4A	3	1
8	:15	:52	4	0
9	:16	:5A	4	1

This list designates the standard locations assumed by BASIC-3. Total base page locations required by the program are:

:00 - :01	Power-up interrupt locations
:02 - :09	User #1 I/O interrupt sequence
:0A	Contains highest address to be used as working storage
:18 - :19	Real Time Clock interrupt locations
:1C	Power Down interrupt location
:22 - :29	User #2 I/O interrupt sequence
:2A - :31	User #3 I/O interrupt sequence
:32 - :39	User #4 I/O interrupt sequence
:3A - :41	User #5 I/O interrupt sequence
:42 - :49	User #6 I/O interrupt sequence
:4A - :51	User #7 I/O interrupt sequence
:52 - :59	User #8 I/O interrupt sequence
:5A - :61	User #9 I/O interrupt sequence
:64 - :FD	BASIC-3 Processor Address Pointers

8.1.3.4 Limitations on User Machine Language Subroutines Using BASIC-3

A machine language subroutine loaded in conjunction with the BASIC-3 object tapes must observe the following limitations:

1. The subroutine should avoid using the base page, as it is required by the BASIC-3 processor and monitor for pointers and interrupt location routines.



NOTE

Base page locations can be made available to machine language subroutines at the expense of one or more users, starting with user # 9. For example, base page locations : 5A through : 61 are used by MONITOR as the I/O interrupt area for user #9. This area can be made available to the LAMBDA loader to use for address pointers if user #9 is not selected. (Note that the LAMBDA loader will decrement the base page address passed to it (: 63 when loading BASIC-3) downward toward location : 00 as it requires a new base page pointer.) Since each user is allocated eight base page locations for its I/O interrupt area, eight additional locations can be made available to LAMBDA for every user (from user #9 down to user #2) which is not selected.

WARNING

Once used by LAMBDA, a user's interrupt area may not later be re-assigned to that user without reloading the BASIC system.

2. The return address of any Jump-and-Store (JST) instructions between the subroutine and the BASIC-3 processor should be saved by calls to PSH: and POP: . If these sequences, or calls to IOC: , are used in the subroutine, the subroutine is required to be re-entrant, as the POP: routine (and the IOC: routine) will allow the next user in turn to take control of BASIC-3 (and the machine language routine) if the current user's time slice is exhausted.
3. If the subroutine does not call either POP: or IOC: , no swapping of users can take place, thus making it possible for all other users to "hang-up" waiting for the subroutine to complete. For this reason, it is advisable that the subroutine be as short as possible.



Appendix A

MACHINE LANGUAGE SUBROUTINES

A.1 INTRODUCTION

The BASIC-2 and BASIC-3 systems may be augmented by user written machine language subroutines which can be referenced by the CALL statement (section 4.3.3). Such subroutines are written in ALPHA-16 assembly language and linked with the BASIC-2 or BASIC-3 system prior to execution of the BASIC program. Refer to Section 8 - OPERATING PROCEDURES for a more detailed discussion of the linkage procedure.

External references to multiple machine language subroutines, labeled ML1 through ML20, exist in the BASIC-2 and BASIC-3 processors. During execution of a BASIC program, the integerized value of the first argument of the CALL statement determines the machine language subroutine executed. For example, if the first argument of a particular CALL statement truncates to the integer 2, the machine language subroutine ML2 will be executed; a value of 4 will cause ML4 to be executed; and so on. A CALL to a non-existent subroutine will cause the "CA" error message to be produced and the BASIC system to return to the command mode.

A.2 CODING THE MACHINE LANGUAGE SUBROUTINE

A subroutine which is to be CALLED by the BASIC-2 or BASIC-3 system must be written in the ALPHA-16 machine language and assembled with the BETA or OMEGA assembler prior to linking with the BASIC system.

The machine language subroutine must have each entry point defined on a NAM directive, which corresponds to an existing BASIC external name (ML1 thru ML20). Conversely, internal BASIC accumulators or routines must be defined as external by use of the EXTR or REF directives.

NOTE

The BASIC-3 system normally does not allow base page usage by the subroutine, and thus the EXTR directive should not be used. Refer to section 8.1.3.4 for limitations on base page usage.

Each entry point should begin with an initial scratch word for storage of the P-register (the CALL statement executes a jump-and-store instruction when entering the subroutine) followed immediately with a call to the BASIC routine PSH: to save the return



address. For example:

	NAM	ML2
	EXTR	PSH:
ML2	ENT	
	JST	PSH:

Upon entry, any arguments which were present in the CALL statement are placed on the BASIC operand stack and are available to the subroutine, in reverse order. That is, if the CALL statement was

CALL (2,A,B,C)

then expression C would be the top (first available) operand on the operand stack, expression B would be below it (available second), and expression A below it (available third).

The subroutine should verify the correct number of arguments supplied in the CALL statement by a call to the BASIC routine VAC: . All arguments passed in the CALL statement must be removed from the operand stack prior to returning to the BASIC processor.

When processing is complete, the subroutine returns to the BASIC processor with a jump to the POP: routine. For example:

EXTR	POP:
•	
•	
JMP	POP:

NOTE

The BASIC-3 system requires the machine-language subroutine to be re-entrant. See section 8.1.3.4 for a discussion of this and other limitations).

A.3 USING THE INTERNAL BASIC ROUTINES

The BASIC-2 and BASIC-3 processors contain internal routines which perform arithmetic operations, operand processing, data manipulation and other related operations. Several of these routines are of value to the machine language subroutine and are described in detail in this section.

A.3.1 BASIC Variables

The BASIC processor's working accumulators are available to the subroutine and perform arithmetic operations and argument transfers. These working accumulators are located in the scratch pad area of memory, and are addressable by use of the EXTR directive.



A.3.1.1 ACC,BCC,CCC

These variables are used in pairs as floating point accumulators. The ACC accumulator consists of two words, labeled ACC1 and ACC2; BCC and CCC are similarly labeled. ACC1 ordinarily contains the first word of a floating point value (the one which contains the mantissa) and ACC2 contains the second word (the mantissa extension). The BCC and CCC accumulators are similarly used.

As a rule, before a new value is placed in the ACC accumulator, the contents of the BCC accumulator are moved to the CCC accumulator and the old contents of the ACC accumulator are moved to the BCC accumulator. This can be accomplished by a call to the STK: routine. Thus, the three double-word accumulators act as a push-down stack.

Conversely, when a floating point value is removed from the ACC accumulator the BCC accumulator is moved to the ACC accumulator and the CCC accumulator is moved to the BCC accumulator. This can be accomplished by a call to the RMV: routine.

A.3.1.2 FROM, TO, NW

These variables serve as arguments to the MOV: (MOVE) routine which is responsible for moving consecutive words from one section of core memory to another.

A.3.1.3 OPDEND

This variable points to the first (lowest) entry in the operand stack.

A.3.2 The BASIC Symbol Table

The BASIC processor maintains a symbol table in core memory which includes all variable symbols defined or referenced in the BASIC program. The symbol table entries are of variable length and new entries are always added at the end. A new entry is added upon the first appearance (during program execution) of every simple variable, array, string variable or DEF function.

A symbol table entry pointer always points to the topmost word of the entry.

The first (top) word of each entry contains its name. The second (next-to-top) word of each entry is the length in words of the entire entry. Thereafter, the layout of an entry depends on its type.



In the following entry layouts, all two-word values are in standard floating-point format with the first word (the one containing the characteristic) occupying the bottom position and the second word (the one containing the mantissa extension) occupying the top position.

Simple Variable Entry

L	letter	digit or space	} name length current value
L-1		4	
L-2	mantissa continued		
L-3	characteristic	mantissa	

The entry for a simple variable used as a FOR loop control has an extended format:

L	letter	digit or space	} name
L-1		9	
L-2	Standard floating-point format		} current value
L-3			
L-4	Standard floating-point format		} limit value
L-5			
L-6	Standard floating-point format		} increment value
L-7			
L-8	return point		

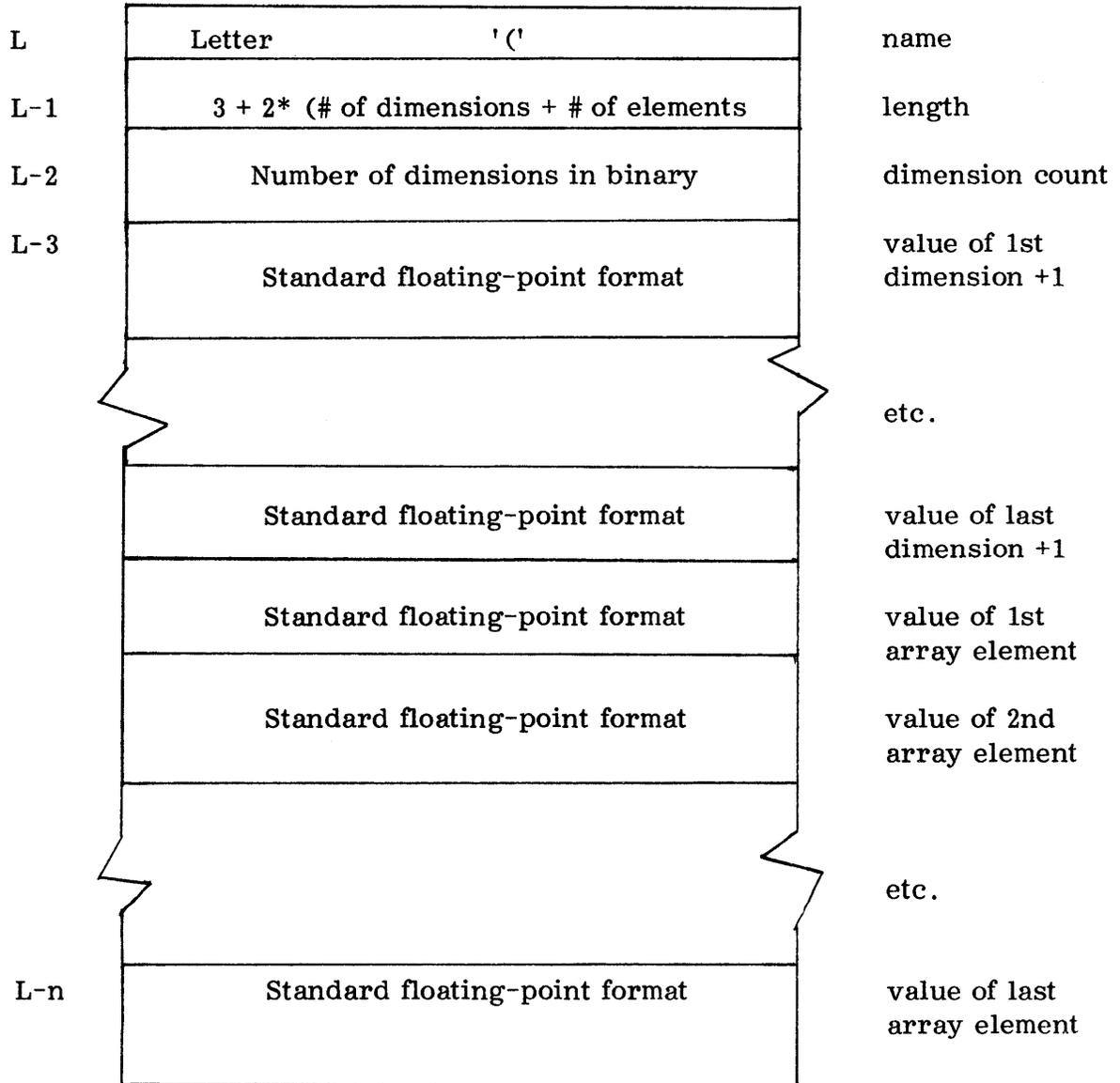


The entry format for a DEF function is:

L	'N'	letter	name
L-1	4 + number of parameters		length
L-2	PNTR value of equals sign		location of function formula
L-3	BASE value of DEF statement		
L-4	letter	digit or space	name of 1st param.
L-5	letter	digit or space	name of 2nd param.
	}		etc.
L-n	letter	digit or space	name of last param.



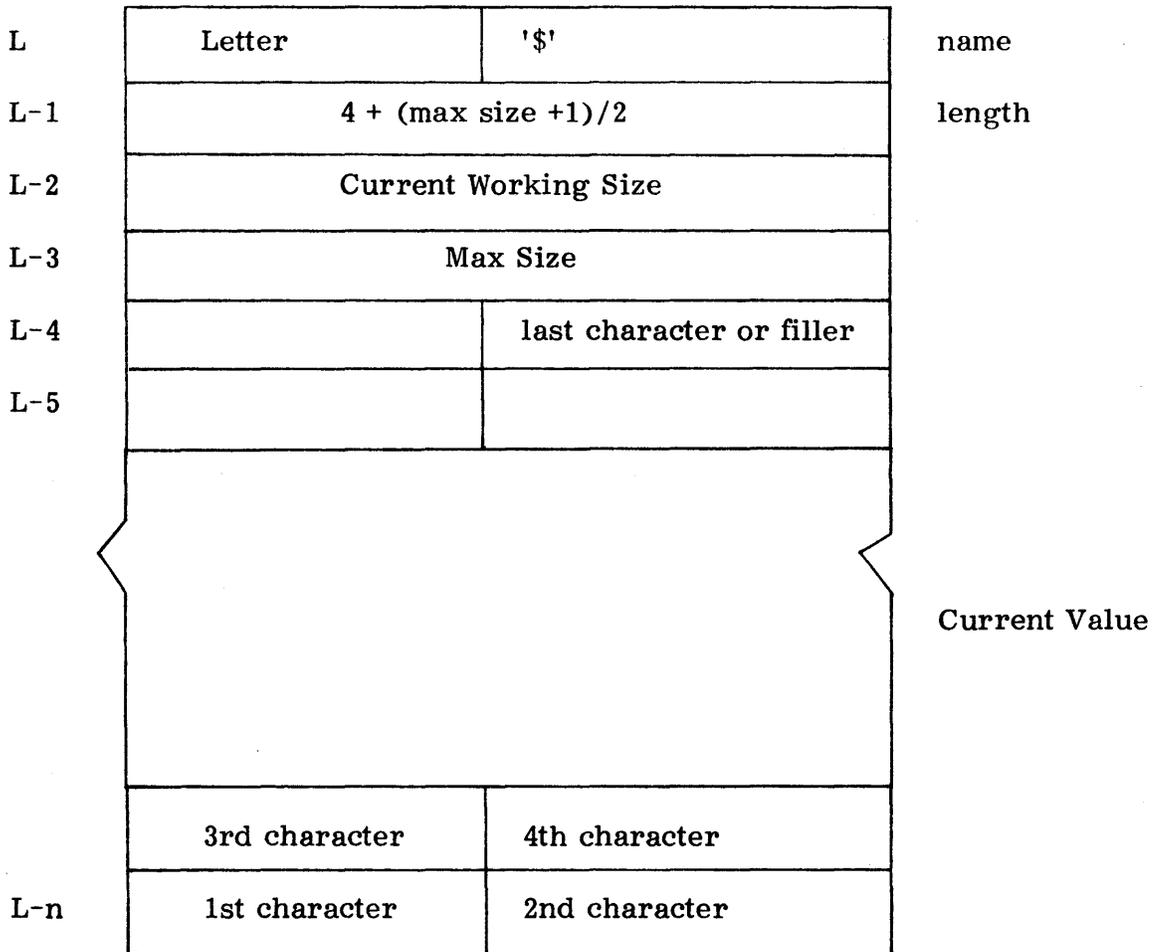
The entry format for an array is:





Array elements are stored in the entry in row-wise order, that is, the topmost element has the subscript (0,0,0,...,0) and the next lower element has the subscript (1,0,0,...,0).

The entry format for a string variable is:



The word size is used here to mean a binary character count. Max size is the value specified by the user in the DIM statement for the string variable.



A.3.3 The BASIC Operand Stack

The operand stack is used during the evaluation of expressions in the BASIC program to save operands that have not yet been processed. It is a last-in, first-out stack with new operands added and old operands removed from the top. The external label OPDEND contains the address of the first (top) entry in the operand stack.

Every numeric operand entry in the stack is two words long. It is either a standard floating-point value or a symbol table pointer. Note that it is possible to distinguish these two cases by the following rule:

If the second word is zero, or if both words are non-zero the operand is a floating-point value; otherwise, the second word contains a word pointer to a symbol table entry.

The format of a floating-point value is described in section 7.3.4.

String operand entries in the stack are of variable length and of three types: string variable, partial string, and string constant. If the top word in the stack is zero, then the operand is a string constant whose length in bytes is given by the next-to-top word. For example, the string constant "PAX" would appear in the operand stack as:

```
: 0000
: 0003
: D8XX
: D0C1
```

where XX represents an unused character position. If the top word is non-zero and the next-to-top word is zero, then the operand is two words long and represents a string variable. The value of the second (top) word is a symbol table pointer, just as in the case of a numeric operand. If both the top two words in the stack are non-zero, the operand is a partial string. In this case the second (top) word is again a symbol table pointer, and the first (next-to-top) word contains the partial string offset in its left byte and the partial string length in its right byte. For example, a source language reference to the partial string S\$(3,10) would result in the following entry in the operand stack:

```
: ssss
: 0208
```

where ssss represents the symbol table pointer for the string variable S.

To summarize the analysis of an entry on the operand stack, let X and Y represent non-zero values, and consider the top words on the stack:



<u>Word 1</u>	<u>Word 2</u>	
0	0	This represents the floating-point value zero.
0	X	This is a symbol table pointer. To distinguish whether it represents a numeric variable or a string variable it is necessary to look at the symbol table entry pointed to by X (*X).
X	0	This is either a string constant or a floating-point value. If bits 15 and 7 of X are both zero it is a string constant whose length in bytes is contained in bits 6 through 0 of X, and the text of the constant is contained in the next (X+1)/2 lower words in the stack. If either bit 15 or 7 of X is on, the operand is a floating-point value.
X	Y	This is either a partial string or a floating-point value. If either bit 15 or 7 of X is on, the operand is a floating-point value. Otherwise, it is a partial string and bits 14 through 8 contain the string offset, bits 6 through 0 contain the length, and Y is a symbol table pointer to the string variable.

This representation of operands in the operand stack makes use of the fact that a) bits 15 and 7 of the first word of a floating-point value are never both zero unless the value is identically zero, and that b) both the offset and length of a string can never exceed 72.

A.3.4 The Internal Floating Point Format

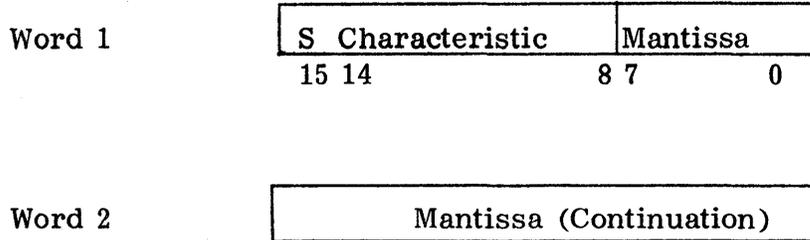
The value of an operand is maintained internally in a floating point format occupying an adjacent pair of computer words.

Bit 15 of the first word indicates the sign of the value (0 = positive, 1 = negative). This is followed by a 7-bit biased binary exponent (characteristic) in the range 0 - 127 (:00 - :7F). The bias is 64, so that the actual range of the characteristic is -64 thru +63.

The characteristic is followed by a normalized 24-bit binary fraction (mantissa) whose implied binary point is to the left of bit 7 of the first word. The value of the mantissa is in the range of 0.5 thru $1 - 5^{-24}$.



The value of zero is represented by a double-word of all zeros. A negative value is represented by taking the two's complement of the left (high-order) word of the floating point representation of the corresponding positive value as defined above.



A.3.5 The BASIC Utility Subroutine

All utility subroutines available in the BASIC processor are called by means of a JST instruction, and all return control to the location immediately following that instruction unless an error is detected. In many cases, the A register is used to pass arguments to/from the subroutines.

Several of the utility routines use a symbol table pointer either to access the current value in the symbol table entry or to store a new value there. They all make the assumption that the value is two words long and is located two words below the word whose address is in the pointer. For example, if the value is located at N-3 and N-2, with N-3 containing the high-order word of the two-word quantity. N normally points to the first word (the one containing the name) of a standard 4-word symbol table entry. However, this is not a requirement. If it is desired, for example, to reference an arbitrary element of an array in the symbol table, all that is necessary is to set N to 3 plus the address of the word containing the characteristic. The reference need not even be in the symbol table. For example, if N is set to the address of ACC1 + 3 then the floating-point value in ACC1,2 will be referenced. Symbol table pointers that do not actually point to the first word of a symbol table entry all called pseudo symbol table pointers.

A.3.5.1 Error Message Print (ERR:)

The ERR: routine will print the two character error messages on the user's console and force control back to the command mode. ERR: is called with a JST, followed immediately by the two ASCII characters to be printed; i.e.,

JST	ERR:
DATA	'ML'



A.3.5.2 Evaluate Operand (EVL:)

EVL: removes the top operand from the operand stack and places its value in the ACC accumulator, after having first moved the contents of the BCC accumulator to the CCC accumulator and the previous contents of the ACC accumulator to the BCC accumulator.

A.3.5.3 Input/Output Control (IOC:)

The IOC: subroutine handles all transfers between the BASIC processor and the user's terminal. Each call to IOC: transfers a sequence of one or more characters between the buffer and the terminal.

The BASIC processor uses the A and X registers to transfer information to the supervisor when it makes an I/O call. This information is not returned in the registers when returning from IOC: .

The A register contains the word address of the buffer from which or into which data is to be moved. This is always an address within the user's work area. If bit 15 of the X register is on, then data is to be transferred to or from paper tape at the user's terminal; if bit 15 of the X register is off, then data is to be transferred to the printer or from the keyboard of the user's terminal. If bit 14 of the X register is on, then the call is for input and bits 7 through 0 contain a prompt character (in ASCII) that is to be printed at the user's terminal before a maximum of 72 bytes are transferred from the terminal into the buffer. (The printing of the prompt character is bypassed if bit 15 of the X register is on.) If bit 14 of the X register is off, then the call is for output and bits 7 through 0 contain a binary count of the number of bytes to be transferred from the buffer to the terminal.

NOTE

In the BASIC-3 system, if a prompt character is to be output prior to an input, it must be issued as a separate output call to IOC: . A four-byte buffer is required, containing the following data:

: 8D : 8A : XX : 00

where : XX represents the 8-bit hex equivalent of the prompt character. The length passed to IOC is 4 bytes.

For an output operation, no manipulation of the data is required; all necessary carriage control characters are already provided in the buffer, and IOC: is not expected to supply any automatically.

For input operations, some character-by-character editing is required. The left-arrow character should be recognized as a backspace signal and the line-feed character should be recognized as a line-erase signal. Normal termination of the input line



is signaled by the appearance of a carriage-return character. The number of characters placed in the buffer (excluding the terminal carriage-return and any trailing blank characters) should be made available in the A register when control is returned to the BASIC processor.

For all I/O operations, characters are packed in the buffer two characters per word, and the internal representation is as specified in Table A-7 of the ALPHA-16 Computer Reference Manual.

A.3.5.4 Move (MOV:)

The MOV: subroutine moves consecutive words from one region of memory into consecutive locations in another region of memory. Prior to calling MOV:, the variables FROM and TO must be initialized to the lowest addresses of the sending and receiving fields, and NW must contain the count of the number of words to be moved. The sending and receiving fields may overlap.

A.3.5.5 Move Bytes (MVB:)

The MVB: subroutine moves consecutive bytes from one region of memory into consecutive locations in another region of memory. Prior to calling MVB:, the variables FROM and TO must be initialized to the byte addresses of the sending and receiving fields, and NW must contain the count of bytes to be moved.

A.3.5.6 Pointer Test (PTT:)

This subroutine determines if the top operand on the operand stack is a value or a symbol table pointer. PTT: returns the A-register equal to zero if it is a symbol table pointer and the A-register equal to -1 if a value.

A.3.5.7 Remove (RMV:)

The contents of the BCC accumulator are moved to the ACC accumulator and the contents of the CCC accumulator are moved to the BCC accumulator.

A.3.5.8 Retrieve (RTV:)

RTV: is called to retrieve the value of a symbol. Upon entry to RTV:, the A-register must contain a pointer to a true or pseudo symbol table. The value of the symbol pointed to is placed into the ACC accumulator after first moving the contents of the BCC accumulator to the CCC accumulator and the previous contents of the ACC accumulator to the BCC accumulator.



A.5.3.9 Stack (STK:)

This subroutine pushes down the entries in the working accumulator. The contents of the BCC accumulator are moved to the CCC accumulator and the contents of the ACC accumulator are moved to the BCC accumulator.

A.3.5.10 Store (STR:)

The STR: subroutine stores the floating point number located in the ACC accumulator into a symbol table entry. Upon entry to STR: the A-register must contain a pointer to a real or pseudo symbol table entry. The contents of the ACC accumulator are then moved to the value positions of this entry.

A.3.5.11 Verify Argument Count (VAC:)

This subroutine verifies the number of arguments passed (and on the operand stack) is correct. Upon entry to VAC:, the A-register must contain a binary count of the total number of fields (arguments plus subroutine number) expected in the CALL statement. VAC: will return if the correct number of arguments exist, or issue an "FA" error message if they do not.

A.3.6 The BASIC Arithmetic Subroutine

The BASIC arithmetic subroutines provide Floating Point arithmetic operations to the calling program. The floating point routines are called by means of a JST instruction and all return control to the location immediately following that instruction, unless an error is detected. Depending upon the subroutine called, the A-register, the ACC accumulator and the BCC accumulator may be used for passing arguments and/or results.

A.3.6.1 Fix (FIX:)

The FIX: subroutine converts the floating point number located in the ACC accumulator to a signed 16-bit binary integer that is returned in the A-register and ACC1 (word 1 of the ACC accumulator). FIX: truncates any excess fractional bits and returns a negative number in its 2's complement form. A number which can not be represented in 15 bits will cause an "NO" error.

A.3.6.2 Floating Add (FAD:)

The FAD: subroutine adds the floating point number in the BCC accumulator to the floating point number in the ACC accumulator, leaving the result in the ACC accumulator. The contents of the CCC accumulator then replace the contents of



the BCC accumulator and FAD: exits. An "OV" error will be generated if the magnitude of the result exceeds the maximum floating point number.

A.3.6.3 Floating Subtract (FSU:)

The FSU: subroutine subtracts the floating point number in the BCC accumulator from the floating point number in the ACC accumulator, leaving the result in the ACC accumulator. The contents of the CCC accumulator are then moved to the BCC accumulator and FSU: exits. An "OV" error will be generated if the magnitude of the result exceeds the maximum floating point number.

A.3.6.4 Floating Multiply (FMU:)

The FMU: subroutine multiplies the floating point number in the ACC accumulator by the floating point number in the BCC accumulator, leaving the result in the ACC accumulator. The contents of the CCC accumulator are then moved to the BCC accumulator and FMU: exits. An "OV" error will be generated if the magnitude of the result exceeds the maximum floating point number.

A.3.6.5 Floating Divide (FDV:)

The FDV: subroutine divides the floating point number in the ACC accumulator by the floating point number in the BCC accumulator, leaving the result in the ACC accumulator. The contents of the CCC accumulator are then moved to the BCCJ accumulator and FDV: exits. An "OV" error will be generated if the magnitude of the result exceeds the maximum floating point number.



Appendix B - BASIC-2 'CALL' EXAMPLE

ALPHA-16 BASIC-2

*TAPE

*LIST

10 REM BASIC 'CALL' STATEMENT EXAMPLE

20 LET A=B=C=D=E=F=0

30 REM ASK USER FOR TWO VALUES

40 INPUT A,B

50 PRINT "PASSED: ";A;B;C;D;E;F

60 REM PASS VALUES (A,B) AND VARIABLES (C,D,E,F)

70 REM TO SUBROUTINE

80 CALL (2,A,B,C,D,E,F)

90 PRINT "EXPECTED: ";A;B;A+B;A-B;A*B;A/B

100 PRINT "RETURNED: ";A;B;C;D;E;F

110 STOP

120 END

*RUN

72,4

PASSED: 2 4 0 0 0 0

EXPECTED: 2 4 6 -2 8 0.5

RETURNED: 2 4 6 -2 8 0.5

*

PAGE 0001

* BASIC "CALL" STATEMENT EXAMPLE

*
* NAM ML2
* EXTR PSH:,VAC:,PTT:,EVL:,STK:,STR:,ERR:,POP:
* EXTR FAD:,FSU:,FMU:,FDV:,OPDEND,ACC1,ACC2

*
* THIS SUBROUTINE EXPECTS TO BE CALLED
* BY THE BASIC PROCESSOR WITH A "CALL"
* STATEMENT OF THE FORM:

* CALL (N,ARG1,ARG2,SYM1,SYM2,SYM3,SYM4)

* THIS SUBROUTINE WILL PERFORM FLOATING
* POINT OPERATIONS AND RETURN THE RESULTS
* IN THE PASSED SYMBOLS AS FOLLOWS:

*
* SYM1 = ARG1 + ARG2
* SYM2 = ARG1 - ARG2
* SYM3 = ARG1 * ARG2
* SYM4 = ARG1 / ARG2

0024
0025 0000 0800
0026 0001 F900
0027 0002 C607
0028 0003 F900
0029
0030
0031
0032
0033
0034
0035
0036 0004 C704
0037 0005 9A2B
0038 0006 B22C
0039 0007 9A2A
0040 0008 F900
0041 0009 3125
0042 000A F900
0043 000B E100
0044 000C B401
0045 000D 9B24
0046 000E DA23
0047 000F DA21
0048 0010 F608
0049
0050
0051
0052 0011 F900
0053 0012 F900
0054

*
* ML2 ENT
* JST PSH: SAVE RETURN
* LAP 7
* JST VAC: VERIFY PARAMETER COUNT
*
* COLLECT THE SYMBOL TABLE POINTERS OF
* THE SYMBOLS PASSED.
*
* NOTE. PARAMETERS ON THE OPERAND STACK
* ARE RETRIEVED IN REVERSE ORDER.
*
* LAM 4 COLLECT 4 POINTERS
* STA CNT
* LDA ASYM1
* STA WRK
* SYMS JST PTT: VERIFY SYMBOL TABLE POINTER
* JAN ERROR . NO, ERROR
* JST EVL: REMOVE FROM OPERAND STACK
* LDX OPDEND COLLECT SYMBOL TABLE POINTER
* LDA #1
* STA *WRK
* IMS WRK
* IMS CNT COUNT DOWN
* JMP SYMS . NOT END, DO AGAIN
*
* COLLECT THE ARGUMENTS ON THE ACCUMULATORS
*
* JST EVL: ARG1 TO ACC
* JST EVL: ARG1 TO BCC, ARG2 TO ACC
*

PAGE	0002			
0055			*	SAVE FOR CALCULATIONS
0056			*	
0057	0013	B100		LDA ACC1
0058	0014	9A23		STA ARG1
0059	0015	B100		LDA ACC2
0060	0016	9A22		STA ARG1+1
0061	0017	F900		JST STK: ARG2 TO BCC AND CCC
0062			*	
0063			*	SYM1 = ARG1 + ARG2
0064			*	
0065	0018	F900		JST RMV: RESTORE ACC
0066	0019	F900		JST FAD: ACC = ACC + BCC
0067	001A	B21C		LDA SYM1
0068	001B	F900		JST STR: STORE IN SYM1
0069			*	
0070			*	SYM2 = ARG1 - ARG2
0071	001C	FA0C		JST RESTR RESTORE ACC
0072	001D	F900		JST FSU: ACC = ACC - BCC
0073	001E	B217		LDA SYM2
0074	001F	F900		JST STR: STORE IN SYM2
0075			*	
0076			*	SYM3 = ARG1 * ARG2
0077			*	
0078	0020	FA08		JST RESTR RESTORE ACC
0079	0021	F900		JST FMU: ACC = ACC * BCC
0080	0022	B212		LDA SYM3
0081	0023	F900		JST STR: STORE IN SYM3
0082			*	
0083			*	SYM4 = ARG1 / ARG2
0084			*	
0085	0024	FA04		JST RESTR RESTORE ACC
0086	0025	F900		JST FDV: ACC = ACC / BCC
0087	0026	B20D		LDA SYM4
0088	0027	F900		JST STR: STORE IN SYM4
0089			*	
0090			*	RETURN TO BASIC
0091			*	
0092	0028	F100		JMP POP:

PAGE 0003

0094

0095

0096

0097 0029 0800

0098 002A B20D

0099 002B 9900

0100 002C B20C

0101 002D 9900

0102 002E F705

0103

0104

0105

0106 002F F900

0107 0030 C0CC

0108

0109 0031 0000

0110 0032 0000

0111 0033 0034

0112 0034 0000

0113 0035 0000

0114 0036 0000

0115 0037 0000

0116 0038 0000

0039 0000

0117

0000 ERRORS

*

*

RETORE ACC WITH ARG1

*

RESTR ENT

LDA ARG1

STA ACC1

LDA ARG1+1

STA ACC2

RTN RESTR

*

*

ISSUE ERROR MESSAGE

*

ERROR JST ERR:

PRINT A MESSAGE

DATA 'ML'

, WHICH IS NOT A BASIC MSG

*

CNT DATA 0

WORK CELL

WRK DATA 0

WORK CELL

ASYM1 DATA SYM4

SYM4 DATA 0

SYMBOL TABLE POINTER STORAGE

SYM3 DATA 0

SYM2 DATA 0

SYM1 DATA 0

ARG1 DATA 0,0

ARG1 STORAGE

END

PAGE 0004

X ACC1 0000

X ACC2 0000

ARG1 0038

ASYM1 0033

CNT 0031

ERROR 002F

X ERR: 0000

X EVL: 0000

X FAD: 0000

X FDV: 0000

X FMU: 0000

X FSU: 0000

X ML2 0000

X OPDEND 0000

X POP: 0000

X PSH: 0000

X PTT: 0000

RESTR 0029

X RMV: 0000

X STK: 0000

X STR: 0000

SYMS 0008

SYM1 0037

SYM2 0036

SYM3 0035

SYM4 0034

X VAC: 0000

WRK 0032

Appendix C - BASIC-2 IOC: SUBROUTINE LISTING

```

PAGE      0001
0001      *      ALPHA-16 BASIC I/O MODULE (IOC:) (96500-1160)
0002      *      COPYRIGHT 1972 BY COMPUTER AUTOMATION, INC.
0003      *      11/17/72
0004      *
0005      *      NAM      IOC:
0006      *      EXTR BUFFER, BASE, PNTR, ERR:
0007      *
0008      ****      THE IOC SUBROUTINE HANDLES ALL I/O
0009      *      OPERATIONS TO OR FROM THE USER'S TERMINAL.
0010      *      UPON ENTRY THE A REGISTER CONTAINS THE
0011      *      WORD ADDRESS OF THE BUFFER INTO OR FROM
0012      *      WHICH DATA IS TO BE TRANSFERRED.  THE X
0013      *      REGISTER CONTAINS A BIT INDICATING WHETHER
0014      *      THE OPERATION IS PAPER TAPE OR NOT, ANOTHER
0015      *      BIT INDICATING WHETHER THE OPERATION IS
0016      *      INPUT OR OUTPUT, AND EITHER A BYTE COUNT
0017      *      FOR OUTPUT OPERATIONS OR A PROMPT
0018      *      CHARACTER FOR INPUT OPERATIONS.  IOC
0019      *      CONTAINS THE LOGIC NECESSARY FOR RESPONDING
0020      *      TO THE BACKSPACE, LINEFEED, AND CARRIAGE-
0021      *      RETURN CHARACTERS DURING INPUT.
0022      *
0023      *      0007      TTY      EQU      7
0024      *      0006      HSPT     EQU      6
0025      *
0026      *      0000      0000      IOC:      ENT
0027      *      0001      5000      STX      SIG      REMEMBER WHETHER TAPE OR KEYBOARD
0028      *      0002      1050      HLH      1
0029      *      0003      9001      STA      BUF      BYTE ADDRESS OF BUFFER
0030      *      0004      1120      RLX      2
0031      *      0005      2214      J05      IOC6      IF INPUT REQUEST
0032      *
0033      *
0034      *      PROCESS OUTPUT REQUEST
0035      *      0006      1009      ARX      2
0036      *      0007      200E      JXZ      IOC4      BYTE COUNT = 0
0037      *      0008      0500      NXR
0038      *      0009      F0AE      STX      CNT      SAVE NEGATIVE BYTE COUNT
0039      *      000A      0E00      IOC2:      SBM
0040      *      000B      F101      LDAB     *BUF     LOAD CURRENT CHARACTER
0041      *      000C      0F00      SWM
0042      *      000D      E000      LTX      SIG     KEYBOARD OR PAPER TAPE ?
0043      *      000E      1120      LTX      1
0044      *      000F      2202      J05      #+3
0045      *      0010      FA46      JST      OKB     KEYBOARD
0046      *      0011      F201      JMP      #+2
0047      *      0012      F05A      JST      OTT     PAPER TAPE
0048      *      0013      D0B1      IMS      BUF     INCREMENT BYTE ADDRESS
0049      *      0014      D0AE      INS      CNT
0050      *      0015      F000      JMP      IOC2     LOOP UNTIL OUTPUT COMPLETE
0051      *      0016      F715      IOC4:      JMP      *IOC:
0052      *
0053      *      PROCESS INPUT REQUEST
0054      *      0017      C08D      IOC5:      LAP      :8D
0055      *      0018      FA3E      JST      OKB     OUTPUT CARRIAGE RETURN

```

PAGE	0002					
0055	0019	FA3D		JST	OKB	ONCE MORE FOR TIMING
0056	001A	B058	IOC6	LDA	SIG	
0057	001B	2081		JAM	IOC7	IF PAPER TAPE
0058	001C	FA3A		JST	OKB	OUTPUT PROMPT CHARACTER
0059	001D	0110	IOC7	ZAR		
0060	001E	98AE		STA	CNT	INITIALIZE INPUT BYTE INDEX
0061	001F	98AF		STA	PTR	
0062	0020	B080	IOC8	LDA	SIG	
0063	0021	2082		JAM	IOC10	IF INPUT FROM TAPE
0064	0022	FA2D		JST	IKB	INPUT FROM KEYBOARD
0065	0023	F203		JMP	IOC12	
0066	0024	FA38	IOC10	JST	IPT	INPUT FROM PAPER TAPE
0067	0025	C000		CAI	:80	
0068	0026	F502		JMP	IOC10	BYPASS LEADER
0069	0027	C08D	IOC12	CAI	:8D	
0070	0028	F21F		JMP	IOC18	IF CARRIAGE-RETURN
0071	0029	C08A		CAI	:8A	
0072	002A	F6L3		JMP	IOC5	IF LINE-FEED
0073	002B	C0DF		CAI	:DF	
0074	002C	F216		JMP	IOC16	IF LEFT ARROW
0075	002D	C0FF		CAI	:FF	
0076	002E	F60E		JMP	IOC8	IF RUBOUT
0077	002F	E0AE		LDX	CNT	
0078	0030	C0A0		CAI	:A0	
0079	0031	F202		JMP	IOC14	IF SPACE
0080	0032	E8AF		STX	PTR	REMEMBER INDEX OF NON-BLANK CHARACTER
0081	0033	D0AF		INS	PTR	
0082	0034	0E00	IOC14	SEM		
0083	0035	90B1		STAB	*@BUF	STORE CHARACTER IN BUFFER
0084	0036	0F00		SNN		
0085	0037	D8AE		INS	CNT	INCREMENT INPUT CHARACTER INDEX
0086	0038	C649		LAF	73	
0087	0039	90AE		SUB	CNT	
0088	003A	D1DA		JAG	IOC8	IF NO OVERFLOW
0089	003B	R100		LDA	BUFFER	
0090	003C	9900		STA	BASE	
0091	003D	B0AE		LDA	CNT	
0092	003E	9900		STA	PNTR	
0093	003F	E100		LDX	BASE	
0094	0040	9080		STA	00	
0095	0041	F900		JST	ERR	
0096	0042	D8C9		DATA	'X1'	EXCESS INPUT
0097	0043	B0AE	IOC16	LDA	CNT	
0098	0044	21E4		JAL	IOC8	
0099	0045	9000		DAP		
0100	0046	38AE		STA	CNT	
0101	0047	F527		JMP	IOC8	
0102	0048	B080	IOC18	LDA	SIG	
0103	0049	30R3		JAM	IOC20	
0104	004A	C68A		LAF	:8A	
0105	004B	FA0B		JST	OKB	ISSUE LINE FEED
0106	004C	F201		JMP	IOC22	
0107	004D	FA0F	IOC26	JST	IPT	
0108	004E	B0AF	IOC22	LDA	PTR	

```

PAGE 0003
0109 004F F74F RTN IOC:
0110 *
0111 * INPUT FROM KEYBOARD
0112 0050 0800 IKB ENT
0113 0051 4038 SEL TTY, 0 AUTO ECHO
0114 0052 4039 SEL TTY, 1 SELECT KEYBOARD
0115 0053 5939 RDA TTY, 1 INPUT TO A
0116 0054 A226 IOR H80 FORCE 7 BIT ON
0117 0055 403C SEL TTY, 4 TURN OFF AUTO ECHO
0118 0056 F706 RTN IKB
0119 *
0120 * OUTPUT TO KEYBOARD
0121 0057 0800 OKB ENT
0122 0058 403C SEL TTY, 4 INITIALIZE TTY
0123 0059 603B WRA TTY, 3 OUTPUT
0124 005A 493B SEN TTY, 3 WAIT UNTIL DONE
0125 005B F601 JMP #-1
0126 005C F705 RTN OKB EXIT
0127 *
0128 * INPUT PAPER TAPE
0129 005D 0800 IPT ENT
0130 005E 5801 ISA INPUT DATA SWITCHES
0131 005F C001 CAI 1
0132 0060 F206 JMP HSR . 1 = READ FROM HSPT
0133 *
0134 0061 403A IPT1 SEL TTY, 2 STEP TTY READER
0135 0062 4839 IPT2 SSN TTY, 1 BUFFER READY?
0136 0063 F206 JMP IPT3 . YES
0137 0064 0150 IAR
0138 0065 3143 JAN IPT2 TRY BUFFER
0139 0066 F605 JMP IPT1 RETRY READER
0140 *
0141 0067 4032 HSR SEL HSPT, 2 STEP HSPT READER
0142 0068 5931 RDA HSPT, 1 INPUT TO A
0143 0069 F201 JMP IPT4
0144 006A 5939 IPT3 RDA TTY, 1 INPUT TO A
0145 006B A20F IPT4 IOR H80
0146 006C F70F RTN IPT
0147 *
0148 * OUTPUT TO PAPER TAPE
0149 006D 0800 OTT ENT
0150 006E 5801 ISX CHECK DATA SWITCHES
0151 006F C101 CXI 1
0152 0070 F205 JMP OTTPT . 1 = PUNCH ON HSPT PUNCH
0153 0071 403C SEL TTY, 4 INITIALIZE TTY
0154 0072 6039 WRA TTY, 1
0155 0073 4939 SEN TTY, 1
0156 0074 F601 JMP #-1
0157 0075 F708 RTN OTT
0158 *
0159 0076 4034 OTIPT SEL HSPT, 4 INITIALIZE HSPT
0160 0077 6031 WRA HSPT, 1
0161 0078 4931 SEN HSPT, 1
0162 0079 F601 JMP #-1

```

```

PAGE 0004
0163 007A F700 RTN OTT
0164 *
0165 00AE CNT EQU :AE
0166 00AF PTR EQU CNT+1
0167 00B0 SIG EQU PTR+1
0168 00B1 BUF EQU SIG+1
0169 007B 00B0 H80 DATA :80
0170 END
0000 ERRORS

```

```

PAGE 0005
X BASE 0000 X BUFFER 0000 BUF 00B1 CNT 00AE
X ERR: 0000 HSPT 0006 HSR 0067 H80 007B
IKR 0050 IOC10 0024 IOC12 0027 IOC14 0034
IOC16 0043 IOC18 0048 IOC2 000A IOC20 004D
IOC22 004E IOC4 0016 IOC5 0017 IOC6 001A
IOC7 001D IOC8 0020 X IOC: 0000 IPT 005D
IPT1 0061 IPT2 0062 IPT3 006A IPT4 006B
OKB 0057 OTTPT 0076 OTT 006D X PNTR 0000
PTR 00AF SIG 00B0 TTY 0007

```

PAGE 0001

X	0000	BASE	0006	0090*	0093					
	0168	BUF	0029*	0039	0047*	0083*				
X	0000	BUFFER	0006	0089						
	0165	CNT	0037*	0048*	0060*	0077	0085*	0087	0091	0097
			0100*	0166						
X	0000	ERR:	0006	0095*						
	0169	H80	0116	0145						
	0024	HSPT	0141	0142	0159	0160	0161			
	0141	HSR	0132							
	0112	IKB	0064*	0118						
	0066	IOC10	0063	0068						
	0069	IOC12	0065							
	0082	IOC14	0079							
	0097	IOC16	0074							
	0102	IOC18	0070							
	0038	IOC2	0049							
	0107	IOC20	0103							
	0108	IOC22	0106							
	0050	IOC4	0035							
	0053	IOC5	0072							
	0056	IOC6	0031							
	0059	IOC7	0057							
	0062	IOC8	0076	0088	0098	0101				
N	0026	IOC:	0005	0050	0109					
	0129	IPT	0066*	0107*	0146					
	0134	IPT1	0139							
	0135	IPT2	0138							
	0144	IPT3	0136							
	0145	IPT4	0143							
	0121	OKB	0044*	0054*	0055*	0058*	0105*	0126		
	0149	OTT	0046*	0157	0163					
	0159	OTTPT	0152							
X	0000	PNTR	0006	0092*						
	0166	PTR	0061*	0080*	0081*	0108	0167			
	0167	SIG	0027*	0041	0056	0062	0102	0168		
	0023	TTY	0113	0114	0115	0117	0122	0123	0124	0134
			0135	0144	0153	0154	0155			

0170 SOURCE LINES