COMPUTER AUTOMATION, INC.

ALPHA-16

BASIC REFERENCE MANUAL

Revised 15 May 1973

Copyright 1973, Computer Automation, Inc.
# TABLE OF CONTENTS

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 PUNCH COMMAND</td>
<td>5-1</td>
</tr>
<tr>
<td>5.4 CLEAR COMMAND</td>
<td>5-2</td>
</tr>
<tr>
<td>5.5 RUN COMMAND</td>
<td>5-2</td>
</tr>
<tr>
<td>5.6 TAPE COMMAND</td>
<td>5-2</td>
</tr>
<tr>
<td>5.7 TYPE COMMAND</td>
<td>5-3</td>
</tr>
</tbody>
</table>

**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:

1. A STOP or END statement is executed
2. An error is detected
3. 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,Δ←←)←(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 \(2.71 \times 10^{-20}\) to \(9.23 \times 10^{18}\). 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Integer</td>
</tr>
<tr>
<td>325</td>
<td>Integer</td>
</tr>
<tr>
<td>2.6</td>
<td>Decimal number</td>
</tr>
<tr>
<td>3.14</td>
<td>Decimal number</td>
</tr>
<tr>
<td>.000314E+4</td>
<td>Decimal number in exponential format</td>
</tr>
<tr>
<td>314E-02</td>
<td>Decimal number in exponential format</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>ARITHMETIC OPERATORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>↑</td>
</tr>
</tbody>
</table>
RELATIONAL OPERATORS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>A&lt; B</td>
<td>A less than B</td>
</tr>
<tr>
<td>&gt;</td>
<td>A &gt; B</td>
<td>A greater than B</td>
</tr>
<tr>
<td>&lt;=</td>
<td>A &lt;= B</td>
<td>A less than or equal to B</td>
</tr>
<tr>
<td>&gt;=</td>
<td>A &gt;= B</td>
<td>A greater than or equal to B</td>
</tr>
<tr>
<td>=</td>
<td>A = B</td>
<td>A equal to B</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>A &lt;&gt; B</td>
<td>A not equal to B</td>
</tr>
</tbody>
</table>

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
- Relationals

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>4</td>
</tr>
<tr>
<td>$Z+7$</td>
<td>4.3</td>
</tr>
<tr>
<td>$-2$</td>
<td>-2</td>
</tr>
<tr>
<td>SGN($Z$)</td>
<td>-1</td>
</tr>
<tr>
<td>INT($Y + \text{ABS}(Z+1)$)</td>
<td>4</td>
</tr>
<tr>
<td>$X \uparrow 2*Z+Y$</td>
<td>-40.06</td>
</tr>
<tr>
<td>$X \uparrow 2*(Z+Y)$</td>
<td>7.04</td>
</tr>
<tr>
<td>$X &lt; Y &lt; Z$</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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 (\(\cdot\)).
- 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:

\[
X \Delta = \Delta \Delta 10.52 \Delta\Delta\Delta\Delta\Delta 10520\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta\Delta -1.052E-02
\]

The deltas (\(\Delta\)) 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(2,2,2,3)
```

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:

\[ \text{MAT} \text{ matrix}=\text{matrix} \]

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:

\[
\text{MAT matrix}=\text{matrix} \times \text{matrix}
\]

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 \( C \):

\[
\text{DIM } A(x,z), B(x,y), C(y,z) \quad \text{MAT } A=B \times C
\]

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:

\[ \text{MAT matrix} = (\text{expression}) \times \text{matrix} \]

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:

\[ \text{MAT matrix} = \text{IDN} \]

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:

\[ \text{MAT matrix} = \text{CON} \]

Effect:

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

4.3.1.11 Print Matrix

General Form:

\[ \text{MAT PRINT matrix [separator matrix] ... separator} \]

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:

\[ \text{MAT READ matrix [,matrix] ...} \]
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:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ\[\]_\`abcdefgijklmnopqrstuvwxyz\{\}|
```

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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>(X)</td>
<td>E raised to the X power</td>
</tr>
<tr>
<td>LOG</td>
<td>(X)</td>
<td>Natural logarithm of X</td>
</tr>
<tr>
<td>ABS</td>
<td>(X)</td>
<td>Absolute value of X</td>
</tr>
<tr>
<td>INT</td>
<td>(X)</td>
<td>The largest integer not greater than X</td>
</tr>
<tr>
<td>MOD</td>
<td>(X,Y)</td>
<td>X modulo Y</td>
</tr>
<tr>
<td>SGN</td>
<td>(X)</td>
<td>A value indicating the sign of X</td>
</tr>
<tr>
<td>SQR</td>
<td>(X)</td>
<td>Square root of X</td>
</tr>
</tbody>
</table>

BASIC-2 and BASIC-3 FUNCTIONS:

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>(X)</td>
<td>Sine of X (X is expressed in radians)</td>
</tr>
<tr>
<td>COS</td>
<td>(X)</td>
<td>Cosine of X (X is expressed in radians)</td>
</tr>
<tr>
<td>TAN</td>
<td>(X)</td>
<td>Tangent of X (X is expressed in radians)</td>
</tr>
<tr>
<td>ATN</td>
<td>(X)</td>
<td>Arctangent of X is returned as an angle in radians in the range ± π/2.</td>
</tr>
<tr>
<td>RND</td>
<td>(X)</td>
<td>A random number between 0 and 1</td>
</tr>
<tr>
<td>LEN</td>
<td>(X$)</td>
<td>Return the length of the string variable X$</td>
</tr>
</tbody>
</table>

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:

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

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

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

<table>
<thead>
<tr>
<th>L</th>
<th>letter</th>
<th>digit or space</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-1</td>
<td></td>
<td>4</td>
<td>length</td>
</tr>
<tr>
<td>L-2</td>
<td>mantissa continued</td>
<td></td>
<td>current value</td>
</tr>
<tr>
<td>L-3</td>
<td>characteristic</td>
<td>mantissa</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>L</th>
<th>letter</th>
<th>digit or space</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-1</td>
<td></td>
<td>9</td>
<td>length</td>
</tr>
<tr>
<td>L-2</td>
<td></td>
<td>Standard floating-point format</td>
<td>current value</td>
</tr>
<tr>
<td>L-3</td>
<td></td>
<td>Standard floating-point format</td>
<td></td>
</tr>
<tr>
<td>L-4</td>
<td></td>
<td>Standard floating-point format</td>
<td></td>
</tr>
<tr>
<td>L-5</td>
<td></td>
<td>Standard floating-point format</td>
<td>limit value</td>
</tr>
<tr>
<td>L-6</td>
<td></td>
<td>Standard floating-point format</td>
<td>increment value</td>
</tr>
<tr>
<td>L-7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-8</td>
<td></td>
<td>return point</td>
<td></td>
</tr>
</tbody>
</table>
The entry format for a DEF function is:

<table>
<thead>
<tr>
<th>L</th>
<th>'N'</th>
<th>letter</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-1</td>
<td></td>
<td>4 + number of parameters</td>
<td>length</td>
</tr>
<tr>
<td>L-2</td>
<td></td>
<td>PNTR value of equals sign</td>
<td>location of</td>
</tr>
<tr>
<td>L-3</td>
<td></td>
<td>BASE value of DEF statement</td>
<td>function formula</td>
</tr>
<tr>
<td>L-4</td>
<td>letter</td>
<td>digit or space</td>
<td>name of 1st param.</td>
</tr>
<tr>
<td>L-5</td>
<td>letter</td>
<td>digit or space</td>
<td>name of 2nd param.</td>
</tr>
<tr>
<td>L-n</td>
<td>letter</td>
<td>digit or space</td>
<td>etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>name of last param.</td>
</tr>
</tbody>
</table>
The entry format for an array is:

<table>
<thead>
<tr>
<th></th>
<th>Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>'C'</td>
<td>name</td>
</tr>
<tr>
<td>L-1</td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>L-2</td>
<td></td>
<td>dimension count</td>
</tr>
<tr>
<td>L-3</td>
<td></td>
<td>value of 1st dimension +1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value of last dimension +1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value of 1st array element</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value of 2nd array element</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc.</td>
</tr>
<tr>
<td>L-n</td>
<td></td>
<td>value of last array element</td>
</tr>
</tbody>
</table>
Array elements are stored in the entry in row-wise order, that is, the topmost element has the subscript \((0,0,0,\ldots,0)\) and the next lower element has the subscript \((1,0,0,\ldots,0)\).

The entry format for a string variable is:

<table>
<thead>
<tr>
<th>L</th>
<th>Letter</th>
<th>'§'</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-1</td>
<td>(4 + (\text{max size} +1)/2)</td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>L-2</td>
<td>Current Working Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-3</td>
<td>Max Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-4</td>
<td>last character or filler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L-5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The word \textit{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:
<table>
<thead>
<tr>
<th>Word 1</th>
<th>Word 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>This represents the floating-point value zero.</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>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).</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Word 1</th>
<th>S Characteristic</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15 14</td>
<td>8 7 0</td>
</tr>
</tbody>
</table>

| Word 2 | Mantissa (Continuation) |

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
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
*
BASIC "CALL" STATEMENT EXAMPLE

* 
NAM ML2
EXTR PSHI, VACI, PTTI, EVLI, STKI, STRI, ERRI, POPI
EXTR FADI, FSUI, FMUI, FDVI, 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

***********************************************************************

025 0000 0800 ML2
026 0001 F900
027 0002 C607
028 0003 F900
029
030
031
032
033
034
035
036 0004 C704
037 0005 9A2B
038 0006 B22C
039 0007 9A2A
040 0008 F900 SYMS
041 0009 3125
042 000A F900
043 000B E100
044 000C B401
045 000D 9B24
046 000E DA23
047 000F DA21
048 0100 F608
049
050
051
052 0011 F900
053 0012 F900
054

ENT
JST PSHI: SAVE RETURN
LAP 7
JST VACI: 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
STA CNT
LDA ASYM1
STA WRK
JST PTTI: 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

B-2
SAVE FOR CALCULATIONS

LDA ACC1
STA ARG1
LDA ACC2
STA ARG1+1
JST STK; ARG2 TO BCC AND CCC

SYM1 = ARG1 + ARG2

JST RMV; RESTORE ACC
JST FAD; ACC = ACC + BCC
LDA SYM1
JST STR; STORE IN SYM1

SYM2 = ARG1 - ARG2

JST RESTR RESTORE ACC
JST FSU; ACC = ACC - BCC
LDA SYM2
JST STR; STORE IN SYM2

SYM3 = ARG1 * ARG2

JST RESTR RESTORE ACC
JST FMU; ACC = ACC * BCC
LDA SYM3
JST STR; STORE IN SYM3

SYM4 = ARG1 / ARG2

JST RESTR RESTORE ACC
JST FDV; ACC = ACC / BCC
LDA SYM4
JST STR; STORE IN SYM4

RETURN TO BASIC

JMP POP1
RESTORE ACC WITH ARG1

LDA ARG1
STA ACC1
LDA ARG1+1
STA ACC2
RTN RESTR

ISSUE ERROR MESSAGE

JST ERR: PRINT A MESSAGE
DATA 'ML': WHICH IS NOT A BASIC MSG

WORK CELL
DATA 0
WORK CELL
DATA 0
SYMBOL TABLE POINTER STORAGE
DATA 0
ARG1 STORAGE
DATA 0,0
END

ARG1 0038
ASYM1 0033
X EVL: 0000
X PSH: 0000
X STK: 0000
X SYM1 0037
X SYM2 0036
X SYM3 0035
X SYM4 0034
X VAC: 0000
WRK 0032
Appendix C - BASIC-2 IOC: SUBROUTINE LISTING

PAGE 0001

0001  * ALPHA-16 BASIC I/O MODULE (IOC:) (96500-1180)
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 0800 IOC:  ENT
0027  0001 E880  STX SIG REMEMBER WHETHER TAPE OR KEYBOARD
0028  0002 1050  MUL 1
0029  0003 98A1  STA BUF BYTE ADDRESS OF BUFFER
0030  0004 1120  RLA 2
0031  0005 3214  JOS IOC6 IF INPUT REQUEST
0032  *
0033  * PROCESS OUTPUT REQUEST
0034  0006 10A9  ARX 2
0035  0007 280E  JX2 IOC4 BYTE COUNT = 0
0036  0008 0508  NXR
0037  0009 E8AE  STX CNT SAVE NEGATIVE BYTE COUNT
0038  000A 0E00 IOC2 SBM
0039  000B 4B11  IDHB *BUF LOAD CURRENT CHARACTER
0040  000C BFA9  SBM
0041  000D E8B9  LIX SIG KEYBOARD OR PAPER TAPE ?
0042  000E 1128  LIX 1
0043  000F 2202  JOS $+3
0044  0010 F446  JST OKB . KEYBOARD
0045  0011 F201  JMP $+2
0046  0012 F455  JST OTT . PAPER TAPE
0047  0013 D8A1  INS BUF INCREMENT BYTE ADDRESS
0048  0014 D8AE  INS CNT
0049  0015 F608  JMP IOC2 LOOP UNTIL OUTPUT COMPLETE
0050  0016 F716 IOC4 JMP *IOC:
0051  *
0052  * PROCESS INPUT REQUEST
0053  0017 08D IOC5 LAR :8D
0054  0018 FA3E JST OKB OUTPUT CARRIAGE RETURN

C-1
<table>
<thead>
<tr>
<th>PAGE</th>
<th>0002</th>
</tr>
</thead>
<tbody>
<tr>
<td>0055</td>
<td>0019</td>
</tr>
<tr>
<td>0056</td>
<td>001A</td>
</tr>
<tr>
<td>0057</td>
<td>001B</td>
</tr>
<tr>
<td>0058</td>
<td>001C</td>
</tr>
<tr>
<td>0059</td>
<td>001D</td>
</tr>
<tr>
<td>0060</td>
<td>001E</td>
</tr>
<tr>
<td>0061</td>
<td>001F</td>
</tr>
<tr>
<td>0062</td>
<td>0020</td>
</tr>
<tr>
<td>0063</td>
<td>0021</td>
</tr>
<tr>
<td>0064</td>
<td>0022</td>
</tr>
<tr>
<td>0065</td>
<td>0023</td>
</tr>
<tr>
<td>0066</td>
<td>0024</td>
</tr>
<tr>
<td>0067</td>
<td>0025</td>
</tr>
<tr>
<td>0068</td>
<td>0026</td>
</tr>
<tr>
<td>0069</td>
<td>0027</td>
</tr>
<tr>
<td>0070</td>
<td>0028</td>
</tr>
<tr>
<td>0071</td>
<td>0029</td>
</tr>
<tr>
<td>0072</td>
<td>002A</td>
</tr>
<tr>
<td>0073</td>
<td>002B</td>
</tr>
<tr>
<td>0074</td>
<td>002C</td>
</tr>
<tr>
<td>0075</td>
<td>002D</td>
</tr>
<tr>
<td>0076</td>
<td>002E</td>
</tr>
<tr>
<td>0077</td>
<td>002F</td>
</tr>
<tr>
<td>0078</td>
<td>0030</td>
</tr>
<tr>
<td>0079</td>
<td>0031</td>
</tr>
<tr>
<td>0080</td>
<td>0032</td>
</tr>
<tr>
<td>0081</td>
<td>0033</td>
</tr>
<tr>
<td>0082</td>
<td>0034</td>
</tr>
<tr>
<td>0083</td>
<td>0035</td>
</tr>
<tr>
<td>0084</td>
<td>0036</td>
</tr>
<tr>
<td>0085</td>
<td>0037</td>
</tr>
<tr>
<td>0086</td>
<td>0038</td>
</tr>
<tr>
<td>0087</td>
<td>0039</td>
</tr>
<tr>
<td>0088</td>
<td>003A</td>
</tr>
<tr>
<td>0089</td>
<td>003B</td>
</tr>
<tr>
<td>0090</td>
<td>003C</td>
</tr>
<tr>
<td>0091</td>
<td>003D</td>
</tr>
<tr>
<td>0092</td>
<td>003E</td>
</tr>
<tr>
<td>0093</td>
<td>003F</td>
</tr>
<tr>
<td>0094</td>
<td>0040</td>
</tr>
<tr>
<td>0095</td>
<td>0041</td>
</tr>
<tr>
<td>0096</td>
<td>0042</td>
</tr>
<tr>
<td>0097</td>
<td>0043</td>
</tr>
<tr>
<td>0098</td>
<td>0044</td>
</tr>
<tr>
<td>0099</td>
<td>0045</td>
</tr>
<tr>
<td>0100</td>
<td>0046</td>
</tr>
<tr>
<td>0101</td>
<td>0047</td>
</tr>
<tr>
<td>0102</td>
<td>0048</td>
</tr>
<tr>
<td>0103</td>
<td>0049</td>
</tr>
<tr>
<td>0104</td>
<td>004A</td>
</tr>
<tr>
<td>0105</td>
<td>004B</td>
</tr>
<tr>
<td>0106</td>
<td>004C</td>
</tr>
<tr>
<td>0107</td>
<td>004D</td>
</tr>
<tr>
<td>0108</td>
<td>004E</td>
</tr>
</tbody>
</table>

C-2
RTN IOC:

INPUT FROM KEYBOARD

SEL TTY, 0 AUTO ECHO
SEL TTY, 1 SELECT KEYBOARD
RDA TTY, 1 INPUT TO A
IOR H80 FORCE 7 BIT ON
SEL TTY, 4 TURN OFF AUTO ECHO
RTN IKB

OUTPUT TO KEYBOARD

SEL TTY, 4 INITIALIZE TTY
WRA TTY, 3 OUTPUT
SEN TTY, 3 WAIT UNTIL DONE
JMP #$-1
RTN OKB EXIT

INPUT PAPER TAPE

ISA INPUT DATA SWITCHES
CAI 1
JMP HSR 1 = READ FROM HSPT

SEL TTY, 2 STEP TTY READER
SSN TTY, 1 BUFFER READY?
JMP IPT3 YES
JAR
JAN IPT2 TRY BUFFER
JMP IPT1 RETRY READER

SEL HSPT, 2 STEP HSPT READER
RDA HSPT, 1 INPUT TO A
JMP IPT4
RDA TTY, 1 INPUT TO A
JMP #$-1
IOR H80
RTN IPT

OUTPUT TO PAPER TAPE

CHECK DATA SWITCHES
CXI 1
JMP OTITP 1 = PUNCH ON HSPT PUNCH
SEL TTY, 4 INITIALIZE TTY
WRA TTY, 1
SEN TTY, 1
JMP #$-1
RTN OTT

SEL HSPT, 4 INITIALIZE HSPT
WRA HSPT, 1
SEN HSPT, 1
JMP #$-1
<table>
<thead>
<tr>
<th>PAGE 0004</th>
</tr>
</thead>
<tbody>
<tr>
<td>0163 007A F70D</td>
</tr>
<tr>
<td>0164</td>
</tr>
<tr>
<td>0165 00AE CNT</td>
</tr>
<tr>
<td>0166 00AF PTR</td>
</tr>
<tr>
<td>0167 0080 SIG</td>
</tr>
<tr>
<td>0168 00B1 BUF</td>
</tr>
<tr>
<td>0169 007B 0030 H80</td>
</tr>
<tr>
<td>0170</td>
</tr>
<tr>
<td>0000 ERRORS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE 0005</th>
</tr>
</thead>
<tbody>
<tr>
<td>X BASE 0000</td>
</tr>
<tr>
<td>X ERR: 0000</td>
</tr>
<tr>
<td>X IOC: 0050</td>
</tr>
<tr>
<td>IOC15 0043</td>
</tr>
<tr>
<td>IOC22 004E</td>
</tr>
<tr>
<td>IOC7 001D</td>
</tr>
<tr>
<td>IPT1 0061</td>
</tr>
<tr>
<td>OLR 0057</td>
</tr>
<tr>
<td>PTR 008F</td>
</tr>
</tbody>
</table>