Series 100 Personal Office Computers
Models 125 and 120

BASIC/125*

*This manual contains information for both Series 100/BASIC and BASIC/125.

HEWLETT PACKARD

978 E. Arques Avenue, Sunnyvale, California 94086

Manual Part No. 45535-90000

Printed in U.S.A.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revised date at the bottom of the page. Note that pages which are rearranged due to changes on an adjacent page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.)

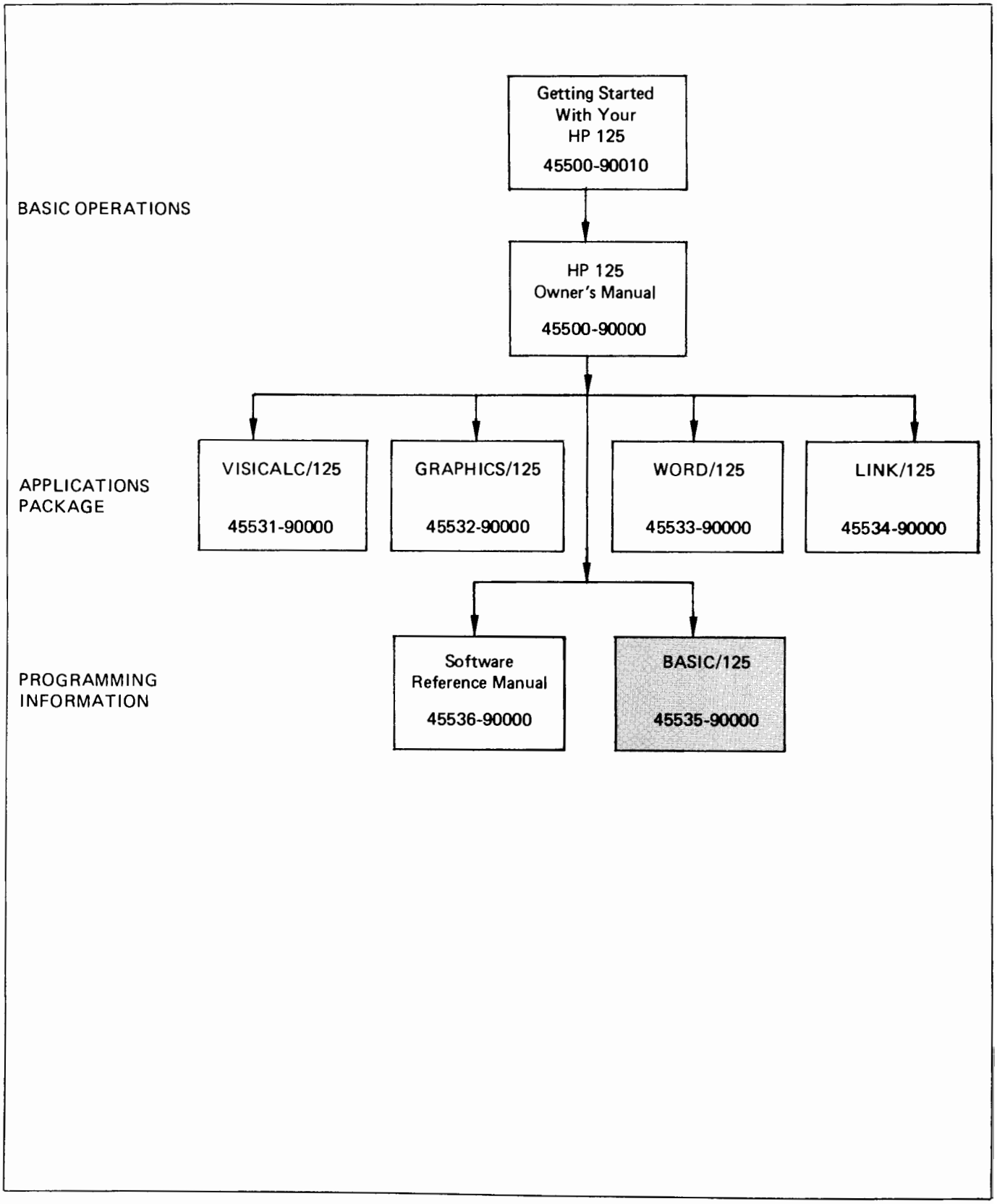
First Edition . . . . . July 1981

Update. . . . . December 1982
pp. 1-2, 1-3, 1-5, 3-26, 3-33, 3-52, 3-64, 3-74, 3-80, 4-24, A-1, F-2, F-3, F-5, H-4, H-5

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Manual Plan


BASIC OPERATIONS
Getting Started With Your HP 125
45500-90010
HP 125 Owner's Manual
45500-90000
APPLICATIONS PACKAGE
VISICALC/125
45531-90000
GRAPHICS/125
45532-90000
WORD/125
45533-90000
LINK/125
45534-90000
PROGRAMMING INFORMATION
Software Reference Manual
45536-90000
BASIC/125
45535-90000

Preface

BASIC/125 is the Hewlett-Packard supported version of Microsoft BASIC-80® , which is a very extensive implementation of BASIC available for the 8080 and Z80 microprocessors. In its fifth major release (Release 5.0), BASIC-80 meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. This manual is a reference for the CP/M® disc version, release 5.0 and later, as supported by Hewlett-Packard Company, on the HP 125 system.

There are significant differences between release 5.0 of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix E for new features in 5.0 that may affect execution.

The manual is divided into five chapters plus a number of appendices.

Chapter 1: General Information
Presents general information about BASIC/125 and includes a section on getting started.

Chapter 2: Data Variables and Operators
Discusses information representation and BASIC/125's built-in operators.

Chapter 3: Statements and Commands
Presents complete information on each of BASIC/125's statements and commands.

Chapter 4: Functions
Provides a detailed description on each of BASIC/125's intrinsic functions.

Chapter 5: Files
Discusses how to create and access disc files.

Appendix A: Error Codes and Error Messages
A detailed list of BASIC/125's error messages and the conditions which cause them.

Appendix B: Reference Tables
Convenient reference tables including ASCII character codes, escape sequences and reserved words.

®CP/M is registered trademark of Digital Research, Inc.
®BASIC-80 is a registered trademark of Microsoft

Table of Contents

Table of Contents (Cont.)

Table of Contents (Cont.)

Chapter 4 BASIC/125 Functions

Table of Contents (Cont.)

Chapter 5 Files

Appendices

CHAPTER 1

GENERAL INFORMATION

Getting Started

The purpose of this section is to explain the use of several fundamental commands which are used to program in BASIC/125. If you haven't already installed y our BASIC/125 software, refer to Appendix G for complete instructions. If you are already familiar with the use of interpretive BASIC then you might wish to skim this section for a quick review.

Each command introduced here will be used in its simplest variation. For complete information on each, see Chapter 3. Commands introduced here are:

BASIC	Loads the BASIC/125 interpreter into memory
AUTO	Generates line numbers automatically after each carriage return. This mode is terminated by typing Ctl-C.
LIST	Lists all or part of a program on the display.
DELETE	Remove a line or lines from a program.
RENUM	Renumbers the lines in a program.
RUN	Executes a program.
SAVE	Saves a copy of a program in a file on disc.
FILES	Lists the names of all the files on the disc.
KILL	Deletes a file from the disc.
NEW	Deletes the program currently in memory so that you can start a new program.
SYSTEM	Exits from BASIC and returns to CP/M.
CTRL-C	Interrrupts execution and returns you to BASIC command level.

Try Writing a Simple Program:

1. Turn on your HP 125 system and bring up the CP/M operating system. (If you don't know how refer to your HP 125 Owner's manual.)

2. When the welcome screen appears, press the BASIC/125 softkey, or press EXIT to CP/M and use the BASIC command as shown below:

```
A>BASIC                <carriage return>
```

When BASIC has loaded into memory, a message similar to this will appear:

```
A>BASIC
BASIC-80 Rev. X.X
[CP/M Version]
Copyright 1977-1981 (C) by Microsoft
Created: dd-mmm-yy
nnnnn Bytes free
OK
```

(X.X represents the revision number for BASIC, dd-mmm-yy are the dates this version was created, and nnnnn represents the number of bytes available in memory for programs and data.)

"OK" is the prompt which tells you that BASIC/125 is ready to accept commands.

3. To start programming type:

```
AUTO                    <carriage return>
```

4. You will automatically be prompted with the line number "10". Type the following program:

```
10 FOR I=1 TO 10                <carriage return>
20    PRINT I;                  <carriage return>
30 NEXT I                       <carriage return>
40 PRINT "LOOP DONE, I=";I      <carriage return>
50 END                          <carriage return>
60 <CTRL>C
```

(Press <CTRL> and C at the same time to stop the line number prompt)

5. To see your program execute type:

```
RUN                     <carriage return>
 1  2  3  4  5  6  7  8  9  10  LOOP DONE, I= 11
OK
```

6. To print a listing of your program on the display type:

```
LIST                    <carriage return>
(System displays this)
10 FOR I=1 TO 10
20    PRINT I;
30 NEXT I
40 PRINT "LOOP DONE, I=";I
50 END
OK
```

7. Delete line 40 from your program by typing:

```
DELETE 40               <carriage return>
```

(NOTE: To change a line in your program you can use the EDIT command, or the Modify Mode softkeys, which are described later in this chapter, or simply retype the line.)

8. LIST your program again and you'll notice that the line numbers are no longer in sequence because you deleted line 40. Renumber your program by typing:

```
RENUM                    <carriage return>
```

If you LIST your program you'll see that the line numbers are once again in sequence.

9. To save your program in a file called PROG1 on disc, type:

```
SAVE "PROG1"             <carriage return>
```

BASIC/125 will supply the CP/M file type .BAS for you. When your file has been written to disc, BASIC will respond with the prompt "OK".

10. To see a listing of the names of all the files on your disc (including the one you just saved), type:

```
FILES                    <carriage return>
```

11. If you wish to erase your program file from disc, type:

```
KILL "PROG1.BAS"    <carriage return>
```

(NOTE: When using the KILL command, the file type (.BAS) must be supplied by you.)

Use the FILES command if you wish to see that your file is gone.

12. To start a new program, type:

```
NEW                      <carriage return>
AUTO                     <carriage return>
```

This removes the previous program from memory, and starts with line number 10 again.

13. When you are ready to exit from BASIC and return to CP/M, simply type:

```
SYSTEM                   <carriage return>
```

(NOTE: Be sure to SAVE your program before exiting, if you wish to use it later.)

Modes of Operation

You may use either of two modes in BASIC: the direct mode or the indirect mode.

In the DIRECT MODE, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The INDIRECT MODE is used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

Line Format

Program lines in a BASIC program have the following format; ([square brackets] indicate optional parts of the statement, <angle brackets> indicate information supplied by you, and nnn represents the line number):

nnnnn <BASIC statement>[:<BASIC statement>...] <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a line of your BASIC program over more than one physical line by typing <CTRL>-J (line feed). This allows you to continue typing a logical line on the next physical line without entering a <carriage return>. Or you can simply keep typing when you reach the end of the line. In this case if WIDTH (see Chapter 3) is left at its default of 72, then a blank line will be inserted in the program listing, but it will not affect program execution. To avoid getting the blank line simply set WIDTH to 255.

Every BASIC program line begins with a LINE NUMBER. Line numbers indicate the order in which program lines are stored in memory and are also used as references in branching and editing. Line numbers must be whole numbers in the range 0 to 65529. A period may be used in the EDIT, LIST, AUTO, and DELETE commands to refer to the current line.

Character Set

The BASIC/125 character set is comprised of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in BASIC/125 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC/125 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC/125:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number or pound sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark or apostrophe
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<DEL>	Deletes the last character typed
<Escape>	Escapes Edit Mode subcommands.
<Backspace>	Backspace and delete character
<Tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<Carriage return>	Terminates input of a line.

The following control characters are recognized by BASIC/l25:

<CTRL>-A	Enters Edit Mode on the line being typed.
<CTRL>-C	Interrupts program execution and returns to BASIC/l25 command level.
<CTRL>-G	Rings the bell at the terminal.
<CTRL>-H	Backspace. Deletes the last character typed. (Note: The easiest way to back-space is to use the Backspace key on the keyboard.)
<CTRL>-I	Tab. Tab stops are every eight columns. (Note: The easiest way to tab is to use the TAB key on the keyboard.)
<CTRL>-J	Line feed.
<CTRL>-O	Halts program output while execution continues.
<CTRL>-R	Retypes the line that is currently being typed, i.e. the newly typed line is "clean", or free from the extraneous characters produced by the DEL key.
<CTRL>-S	Suspends program execution.
<CTRL>-Q	Resumes program execution after <CTRL>-S.
<CTRL>-U	Deletes the line that is currently being typed.

Error Messages

If BASIC/l25 detects an error that causes program execution to terminate, an error message is printed. For a complete list of error codes and error messages, see Appendix A.

Input Editing

To delete a line that is in the process of being typed, type <CTRL>-U. A carriage return is executed automatically after the line is deleted.

To delete the entire program that is currently residing in memory, enter the NEW command (See Chapter 3). NEW is usually used to clear memory prior to entering a new program.

As each character or terminal key is typed at the keyboard it is transmitted to BASIC/125. However, in normal mode Basic/125 only recognizes the terminal keys which are included in the previously described character set. The special HP 125 editing keys (such as DEL CHAR, INS CHAR, etc.) are only recognized by BASIC/125 in MODIFY MODE, which is described below. For example, if you use the cursor control keys to backspace while not in Modify mode, it will appear as though the characters that you backspaced over are gone, while they have actually already been received by BASIC!

If an incorrect character is entered as a line is being typed (while NOT in Modify Mode), it can be deleted with the Backspace key, or with the DEL key, or with <CTRL>-H. DEL surrounds the deleted character(s) with backslashes, and <CTRL>-H has the same effect as backspacing over a character and erasing it. Once a character(s) has been deleted, continue typing the line as desired.

There are two ways to correct program lines for a program that is currently in memory: Modify mode and Edit mode. Or you may simply retype the entire line.

Modify Mode

Modify mode is an easy way to edit program lines without retyping the entire line. Use the cursor control keys to place the cursor anywhere on the display. Position it on the line you wish to edit and enter Modify mode as described below. Make any changes or corrections using the HP 125's character editing keys, and then press the RETURN key to enter the edited line.

The Modify softkeys are accessed by pressing the MODES key on the keyboard. Two MODES softkeys then appear which are used to put the HP 125 into one of two modes:

MODIFY LINE *	When enabled, this mode allows you to edit a line of data, then press the RETURN key to enter the line. When Modify Line is active, an asterisk is present in the softkey label. Modify Line mode automatically ends when the RETURN key is pressed (i.e. it applies to one line only).
MODIFY ALL *	This function is similar to Modify line except that Modify All mode remains on until it is turned off; it is not ended by the RETURN key. When Modify All is active, an asterisk is present in the label. Pressing MODIFY ALL while this mode is active ends the mode and removes the asterisk from the label.

While in Modify Line or Modify All mode you can use the cursor control keys to move the cursor. You can also use the editing keys, INS CHAR and DEL CHAR to insert and delete characters from program lines. To delete a character simply place the cursor at the character you wish to delete and press DEL CHAR. To delete more than one character press DEL CHAR once for each character to be deleted. The INS CHAR key toggles on and off. When it is on the letters IC will appear under the column indicator on the display. Each character you type will be will be inserted at the location of the cursor. To turn off insert character, press INS CHAR a second time.

(NOTE: INS CHAR and DEL CHAR only affect program lines while you are in Modify Line or Modify All mode, at any other time their results will be unpredictable. <CTRL>-C has no effect in modify mode. DO NOT use either of the Modify Modes when the AUTO command is active.)

In Modify Mode, when you press <RETURN>, all characters at and to the right of a pointer get transmitted to BASIC (as though you had typed them very fast). This pointer is the "start of text" pointer (if there is one), or it is the "start column" if no "start of text" pointer exists. Every line in display memory initially does NOT have a "start of text" pointer. A line only acquires a "start of text" pointer under the following conditions:

- The line which you are editing is at the bottom of display memory (last line).

- A character must be typed from the keyboard (not by the computer) on the line. This character must be alphanumeric, a space, a backspace, or a control character. (The cursor control keys have no effect.) The "start of text" pointer then points to the column of this first character.

If no "start of text" pointer exists for a line then the configuration "start column" is used to indicate which column to start the transmission from. To manually set the "start column" indicator, press the AIDS key, and then press the margin/tabs/col key. The START COLUMN key will then be displayed. Move the cursor to the column which is to be the start column and then press the START COLUMN key. To determine what the current "start column" value is, press the AIDS key, and then press the config key. The "start column" may also be changed by positioning the cursor in the Start Column field on the configuration page and using the NEXT CHOICE and PREVIOUS CHOICE softkeys. Press exit to return to BASIC.

One way in which you might wish to use the "start column" feature is to execute a program line in direct mode.

```
10 FOR I=1 TO 10
20 PRINT "THIS IS A TEST"
30 NEXT I

LIST 20                                  (List the line to execute)
20 PRINT "THIS IS A TEST"                (Move the cursor to the P in
THIS IS A TEST                            PRINT using the cursor
OK                                        keys, press AIDS, press
                                          margin/tabs/col, press
                                          START COLUMN, press MODES,
                                          press MODIFY LINE, press
                                          RETURN)
```

For further information on Modify Mode refer to your HP 125's Owner's Manual.

Edit Mode

Edit Mode uses special one-character subcommands to edit a line of your BASIC program. Typically, using Modify Mode with the HP 125 editing keys will be an easier way to edit program lines. However, the Edit Mode commands are included for compatibility with other implementations of BASIC.

Note: Edit mode must be used to modify a BASIC statement that is contained on more than one physical line.

Edit Mode is entered by typing EDIT, and optionally, a line number. Upon entering edit mode, BASIC/125 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search or text within a line. The subcommands are not displayed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

a. Moving the cursor

b. Inserting text

c. Deleting text

d. Finding text

e. Replacing text

f. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape key.

1. Moving the Cursor

<Space> Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

<DEL> In Edit Mode, [i]DEL moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. The ESC key terminates insertion. If <carriage return> is typed during an Insert command, the effect is the same as the ESC key and then <carriage return>. During an Insert command, the DEL key on terminal may be used to delete characters to the left of the cursor. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (<CTRL>-G) is typed and the character is not printed.

X The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, use the ESC key or <carriage return>.

3. Deleting Text

D [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, [i]D deletes the remainder of the line.

H H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of the line.

4. Finding Text

S The subcommand [i]S<ch> searches for the ith occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Pressing <carriage Return> prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC/l25 command level, WITHOUT saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any of the changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC/125 receives an unrecognizable command or illegal character while in Edit Mode, it prints bell (<CTRL>-G) and the command or character is ignored.

Run-time Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC/125 automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC/125 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC/125 will return to command level, and all variable values will be preserved.

<CTRL>-A

To enter Edit Mode on the line you are currently typing, type <CTRL>-A. BASIC/125 responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

Printing Operations

There are two methods used to access a printer from BASIC/125: via BASIC commands and statements, or by using the HP 125's printer control softkeys.

The printer control keys are used mainly during program development. To access them press the AIDS key on the keyboard and then press the printer control softkey (f1). Pressing the printer modes softkey displays a menu which is used for specifying which printer will be used, and for turning on the logging functions. To specify a printer simply locate its softkey label and press the corresponding key on the keyboard (f2-f4). An asterisk is displayed in the label for the printer which will be "controlled" by the softkeys. The logging functions are turned on and off in the same manner. When LOG BOTTOM is on (asterisk in its label), each line in the terminal's memory will be printed as it is added to the bottom of lines which are already displayed. When LOG TOP is turned on (asterisk in its label), lines are only printed as they are lost or "pushed" off the top of the terminal's memory.

Pressing printer control also displays printer control functions which happen immediately, that is, they don't toggle on and off. COPY LINE prints the line which contains the cursor, COPY PAGE prints one page of the terminal's display memory, and COPY ALL prints the contents of the terminal's display memory starting at the location of the cursor and beyond. Use ADVANCE LINE and ADVANCE PAGE to move the printer paper forward without printing anything.

The "L" commands and statements are used to print to the CP/M general list device (as specified in the configuration menu), and are not affected by the printer control softkeys. LLIST prints a program listing directly to the printer. LPRINT and LPRINT USING are used to print information (such as reports) from within a program. To specify a printer (other than the configuration CP/M general list device) from within a program, CP/M's IOBYTE (location 3) must be POKED with a value which specifies the printer as listed below:

```
&HC0     HPIB printer
&H40      Internal thermal printer
&H80       Serial printer
```

CAUTION: Once printing is complete it is necessary to POKE IOBYTE and return it to its original value. To do this use the PEEK function to determine the original value to store. For example:

```
10 BYTE=PEEK (3)                          'STORE ORIGINAL VALUE
20 POKE &H3, &HC0                         'SPECIFY HPIB PRINTER
30 GOSUB 500
40 POKE 3,BYTE                            'RETURN TO ORIGINAL VALUE
.
.
500 LPRINT "ANNUAL SALES REPORT"          'OUTPUT TO PRINTER
```

CHAPTER 2

DATA VARIABLES AND OPERATORS

This chapter discusses data representation and the mathematical and logical operators provided by BASIC/125.

Numeric values may be either single precision, double precision, or integer numbers. All three types are stored in binary representation, integers requiring 2 bytes of memory storage, single precision numbers requiring 4 bytes of memory storage, and double precision numbers requiring 8 bytes. An integer type value may represent a whole number between -32768 and +32767. Single precision numbers have 24 bits of precision, or approximately 7 decimal digits of precision, and are printed with up to 6 decimal digits. Double precision numbers have 56 bits of precision, or approximately 16 decimal digits of precision, and are printed with up to 16 deicmal digits. The internal representations of numeric values is discussed in detail in Appendix C.

Constants

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants are:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants

   Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed point constants

   Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating point constants

   Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10E-38 to 10E+38. Examples:

   ```
   235.988E-7 = .0000235988
   2359E6 = 2359000000
   ```

   (Double precision floating point constants use the letter D instead of E.)

4. Hex constants

   Hexadecimal numbers with the prefix &H. Examples:

   ```
   &H76
   &H32F
   ```

5. Octal constants

   Octal numbers with the prefix &O or &. Examples:

   ```
   &O347
   &1234
   ```

Single and Double Precision Form For Numeric Constants

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Variables

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

Variable Names and Declaration Characters

BASIC/125 variable names may be any length, however, only up to 40 characters are significant. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word, but can contain embedded reserved words. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC/125 commands, statements, function names, and operator names (See Appendix B).

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$="SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single, or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples:

```
PI#       Declares a double precision value
MINIMUM!  Declares a single precision value
LIMIT%    Declares an integer value
```

There is a second method by which variable types may be declared. The BASIC/125 statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 3.

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one dimensional array, T(1,4) would reference a value in a two dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Type Conversion

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.) Example:

```
10 A% = 23.42
20 PRINT A%
RUN
  23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
Examples:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
```

The arithmetic was performed in double precision and the result returned in D# as a double precision value.

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to +32767 or an "overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04     2.039999961853027
```

Expressions and Operators

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC/125 may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y X-Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the above order of operations is maintained.

Here are some algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression	
$X+2Y$	X+Y*2	
$X-\frac{X}{Y}$	X-Y/Z	
$\frac{XY}{Z}$	X*Y/Z	
$\frac{X+Y}{Z}$	(X+Y)/Z	
$(X^2)Y$	(X^2)*Y	
X^{Y^Z}	X^(Y^Z)	
X(-Y)	X*(-Y)	Two consecutive operators must be separated by parentheses.
X(-Y)		

Integer Division and Modulus Arithmetic

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer. For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2  (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

Overflow and Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues. Machine infinity is approximately equal to 1.7*10^38.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Chapter 3.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Chapter 3.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (See IF, Chapter 3). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help to demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 and 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC/125 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC/125's intrinsic functions are described in Chapter 4.

BASIC/125 also allows "user defined" functions that are written by the programmer. See DEF FN, Chapter 3.

String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE"  :  B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparision, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"     where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

CHAPTER 3

BASIC/125 STATEMENTS AND COMMANDS

All of the BASIC/125 commands and statements are described in detail in this chapter and are listed in alphabetical order for easy reference. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

A quick reference listing, which shows format only, is contained in Appendix F.

Format Notation

Whenever the format for a statement or command is given, the following rules apply:

1. Items in CAPITAL LETTERS must be input as shown. (However, capitalization is used for syntax only, BASIC/125 will automatically upshift lower case variable names and statement names to upper case.)

2. Items in lower case letters enclosed in <angle brackets> are to be supplied by the user.

3. Items in [square brackets] are optional.

4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

AUTO

Format: AUTO [<line number>[,<increment>]]

Purpose: This command, used when you initially enter a program, generates a line number automatically after each carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

The AUTO command is terminated by typing <CTRL>-C. The line in which <CTRL>-C is typed is not saved. After <CTRL>-C is typed, BASIC returns to command level. If the line in which <CTRL>-C is typed has an asterisk after the line number (indicating that the line already exists), the line will remain unchanged and BASIC will return to command level.

Example:

```
AUTO 100,50       Generates line numbers 100,
                   150,200...

AUTO               Generates line numbers 10,
                   20,30,40...
```

BASIC

Format: BASIC

or

BASIC [<filename>][/F:<number of files>][/M:<highest memory location>][/S:<maximum record size>]

Purpose: To load BASIC/125 into memory after bringing up CP/M.

Remarks: To run BASIC, first bring up CP/M (if you don't know how, refer to your HP 125 Owner's manual).

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after BASIC loads into memory. The default file type of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement to return to CP/M when they have finished, allowing the next program in the batch stream to execute.

If /F:<number of files> is present, it sets the number of disc data files that may be open at any time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F: option is omitted. the number of files defaults to 3.

The /M: <highest memory location> option sets the highest memory location that will be used by BASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in memory locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used. For more information on Assembly language subroutines, see Appendix C.

The /S:<maximum record> option may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE: <number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

```
A>BASIC PAYROLL.BAS             Use all memory and 3
                                 files, load and execute
                                 PAYROLL.BAS.

A>BASIC INVENT/F:6              Use all memory and 6
                                 files, load and execute
                                 INVENT.BAS.

A>BASIC /M:32768                Use the first 32K of
                                 memory and 3 files.

A>BASIC DATACK/F:2/M:&H9000     Use the first 36K of
                                 memory, 2 files, and
                                 execute DATACK.BAS.
```

CALL

Format: CALL<variable name>[(<argument list>)]

Purpose: To Call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an assembly language subroutine. It is not normally used in BASIC/l25-only programming. (See also the USR function, Chapter 4)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the assembly language subroutine. <argument list> may not contain literals.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
      .
      .
      .
```

For further information on Assembly language subroutines, see Appendix C.

CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>]
[,ALL][,DELETE<range>]]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

When this statement executes, BASIC/125 will search for the file "PROG1.BAS" on the current disc. If it is found, the program is loaded and executed. If it is not located then a "FILE NOT FOUND" error occurs, and if no ON ERROR statement is active, execution halts and the user is returned to command mode.

To CHAIN to a file which is located on a drive other than the current one, include the CP/M drive specifier of a letter followed by a colon. Example:

```
CHAIN "B:PROG2"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

Because <line number exp> refers to a line in another program, it is not affected by a RENUM command, which only effects lines in the current program.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVERLAY",1000
```

After an overlay is brought in, and is finished being it is usually desirable to delete it so that a new ove may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE "OVRLAY2",1000,DELETE 1000-5000
```

The above statement deletes lines 1000 to 5000 in the current program, merges in the file "OVRLAY2.BAS", and resumes execution at line 1000.

The line numbers in <range> are affected by the RENUM command, since they refer to lines in the current program.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

The Microsoft BASIC compiler does not support the ALL, MERGE, and DELETE options to CHAIN. If you wish to maintain compatibility with the BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used.

CLEAR

Format: CLEAR [[,<expression1>][,<expression2>]]

Purpose: To set all numeric variables to zero and all string variables to null; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC/125. The default is for BASIC to use all of the memory up to the start of CP/M's FDOS.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

Examples:

```
CLEAR

CLEAR ,44261

CLEAR ,,2000

CLEAR ,54261,2000
```

CLOSE

Format: CLOSE[[#]<file number>[,[#[<file number...>]]

Purpose: To conclude I/O to a disc file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disc files automatically. (STOP does not close disc files.)

Example:

```
100 OPEN "O",#2,"OUTFILE"
110 PRINT #2, CNAME$,ADDRESS$,ZIP$,PHONE$
120 CLOSE #2
```

(For further information on files, see Chapter 5.)

COMMON

Format: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. Variables in the main program are passed to variables in the CHAINED program by listing each by its name in a COMMON statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
10 COMMON CUST$,A,F()              (Listing for FIL2)
20 PRINT CUST$,A,F(1)

10 A=10:  CUST$="BOB":  B=20       (Listing for FIL1)
20 COMMON A,CUST$,B
30 CHAIN "FIL2"
RUN
BOB              10              0
OK
```

Notice in the above example that the value for the variable F(1) was printed as 0. This is because a variable whose name is F was not included in the common statement for FIL1, so no value for F(1) was passed to FIL2.

CONT

Format: CONT

Purpose: To continue program execution after a <CTRL>-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break.

Example:

```
10 INPUT "ENTER THE PRICE ",A
20 IF A<20! THEN SURCHG=1!
30 STOP
40 TOT=A+SURCHG
50 PRINT TOT
RUN
ENTER THE PRICE 15          (User types 15)
Break in 30
OK
PRINT SURCHG                (User types this)
 1
OK
CONT                        (User types this)
 16
OK
```

For more information see the STOP command.

DATA

Format: DATA <list of constants>

Purpose: To store the numeric and string constants (i.e. the data) that are accessed by the program's READ statement(s). (See READ)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas) and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement, or else a "TYPE MISMATCH" error will occur.

DATA statements may be reread by use of the RESTORE statement. (For more information, see RESTORE)

Example:

```
70 DATA 88,APRIL,125.32,57271.20
80 READ A,MONTH$,PRICE!,SALES#
90 UNITS=SALES#/PRICE!
100 PRINT MONTH$;" UNIT SALES = ";UNITS
RUN
APRIL UNIT SALES =  457
OK
```

DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only as formal parameters to define the function; they do no affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name (See Chapter Two - Type Declaration), the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Examples: If a program contains -

```
30 VALUE(I)=A+Y/F-D
    .
    .
    .
80 VALUE(I)=B+Y/F-E
    .
    .
    .
200 VALUE(I)=C+Y/F-G
```

By defining a function -

```
10 DEF FNNUM(S,T)=S+Y/F-T
```

The program can be simplified like this -

```
30 VALUE(I)=FNNUM(A,D)
    .
    .
80 VALUE(I)=FNNUM(B,E)
    .
    .
    .
200 VALUE(I)=FNNUM(C,G)
```

The following program uses a string function to underline a string variable on the display, and a numeric function to compute the average of three numbers.

```
60 DEF FNUL$(ST$)=CHR$(27)+"&dD"+ST$+CHR$(27)+"&d@"
70 DEF FNAVG(A,B,C)=(A+B+C)/3
80 INPUT "ENTER NAME AND SCORES ";NM$,L,M,N
90 PRINT FNUL$(NM$)
100 PRINT FNAVG(L,M,N)
RUN
ENTER NAME AND SCORES DAVE,5,8,5    (User response)
DAVE
 6
OK
```

DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Purpose: To declare that certain variable names are to be treated automatically as type integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable. Example:

```
10 DEFDBL B-D
20 D=5.2D+17 : C%=20.2
30 PRINT D,C%
RUN
 5.2D+17          20
OK
```

In the above example the variable C% is printed as an integer because of the type declaration character, %, even though C is in the range list of the DEFDBL declaration.

If no type declaration statements are encountered, BASIC/l25 assumes all variables without declaration characters are single precision varaibles.

Examples:

```
10 DEFDBL L-P   All variables beginning with
                the letters L,M,N,O, and P
                will be double precision
                variables.

10 DEFSTR A     All variables beginning with
                the letter A will be string
                variables.

10 DEFINT I-N,W-Z
                All variable beginning with
                the letters I,J,K,L,M
                N,W,X,Y,Z will be integer
                variables.
```

DEF USR

Format: DEF USR[<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: DEF USR allows the programmer to define starting addresses for user-defined assembly language functions (up to 10 at a time) to be called from BASIC/l25 programs. This statement is used to specify the starting address of a routine, prior to its actual use. The routines are identified as USR0 through USR9.

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C for information about Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
 .
 .
 .
200 DEF USR0=24000
210 X=USR0 (Y^2/2,89)
 .
 .
 .
```

DELETE

Format: DELETE[<line number>][-<line number>]

Purpose: To delete program lines.

Remarks: BASIC/l25 always returns to command level after a DELETE is executed. If <line number> does not exist, an "illegal function call" error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

DIM

Format: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts, allocate storage accordingly, and initialize array values to zero.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0 unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 to 20
30 READ A(I)
40 NEXT I
    .
    .
    .
```

END

Format: END

Purpose: To terminate program excution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional, if it is not present then execution will stop when there are no more program lines to execute.

BASIC/l25 always returns to command level after an END is executed.

Examples:

```
520 IF K>1000 THEN END ELSE GOTO 20
.
.
.
800 GOSUB 900
820 END
.
.
.
900 CLOSE #2
920 RETURN
```

ERASE

Format: ERASE <list of array variables>

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

(NOTE: The Microsoft BASIC compiler does not support the ERASE statement.)

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

ERROR

Format: ERROR <integer expression>

Purpose: 1) To simulate the occurrence of a BASIC/125 error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC/125 (see Appendix A), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC/125's error codes. (It is preferrable to use the highest available values, so compatability may be maintained if more error codes are added to BASIC/125.) This user defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC/125 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
List
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC/125 types this line)
Ok
```

Example 2:

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
 .
 .
 .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS 5000"
410 IF ERL = 130 THEN RESUME 120
 .
 .
 .
```

See Chapter 4 for more information on ERR and ERL.

FIELD

Format: FIELD[#]<file number>,<field width> AS <string variables>.

Purpose: To allocate space for variables in a random file buffer.

Remarks: BASIC/l25 random files are read and written through a file buffer which holds the file record. This buffer must be assembled and disassembled into individual variables. To get data out of a random buffer after a GET, or to enter data before a PUT, the FIELD statement must be executed to specify the layout of the file buffer.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>.
For example,

```
FIELD#1, 20 AS CNAME$, 10 AS ID$, 40 AS ADDRESS$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable CNAME$, the next 10 positions to ID$, and the next 40 positions to ADDRESS$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.) It merely establishes a template for formatting the buffer.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

If the record layout requires more than 255 characters to define, then the FIELD statement will require more than one program line. To do this you will need to use more than one FIELD statement to define the record:

```
10 OPEN "R",#1,"FILE",120
20 FIELD#1,2 AS ACODE$, 2 AS BCODE$, 4 AS ACTNM$, 2 AS
  DCODE$, 6 AS CITY$, 10 AS LASTNAME$, 2 AS ALTCODE$,
  4 AS OPFLAG$, 2 AS KYNUM$, 8 AS BDATE$, 8 AS
  LOANDATE$, 2 AS PAYCODE$, 5 AS PYMTCRD$, 5 AS
  CHECKNUM$
30 FIELD #1, 62 AS DUM$, 40 AS COMMENTS$, 18 AS
  FRSTNAME$
```

DUM$ represents a string variable whose width is equal to the total width of all the variables in in the previous FIELD statement combined. It serves merely to provide a way of skipping over the buffer space which was allocated to variables in the first FIELD statement. Do NOT assign a value (LSET or RSET) to it.

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example:

```
10 OPEN "R",#1,"FILE",40
20 FIELD #1, 20 AS CUST$,4 AS PRICE$,16 AS CITY$
30 INPUT "CUSTOMER NUMBER",CODE%
40 INPUT "CUSTOMER NAME";CNAME$
50 INPUT "TOTAL ORDER";AMT
60 INPUT "CITY";TOWN$
70 LSET CUST$=CNAME$
80 LSET PRICE$=MKS$(AMT)
90 LSET CITY$=TOWN$
100 PUT #1,CODE%
110 GOTO 30
```

See Chapter 4 for information on the MKS$ function. For more information on using files, see Chapter 5.

NOTE: (Do not use a FIELDed variable name in an INPUT or LET statement.) Once a variable name is FIELDed, it points to the current place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

FILES

Format: FILES[<filename>]

Purpose: To print the names of files residing on disc.

Remarks: If <filename> is omitted, all of the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or file type. An asterisk in the filename or file type will match any file or any file type.

Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

FOR...NEXT

Format: FOR <variable>=x TO y [STEP z]

.
.
.

NEXT [<variable>][,<variable>...]

where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC/l25 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. It is allowable to modify the value of <variable> from inside the loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

(NOTE: Previous versions of Microsoft BASIC set the initial value of the variable loop before setting the final value; i.e., the above loop would have executed six times.)

GET

Format: GET [#]<file number>[,<record number>]

Purpose: To read a record from a random disc file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example:

```
10 OPEN "R",#1,"FILE",40
20 FIELD #1, 20 AS CUST$, 4 AS PRICE$, 16 AS CITY$
30 INPUT "ENTER CUSTOMER NUMBER";CODE%
40 GET #1,CODE%
50 PRINT CUST$
60 PRINT USING "$$###.##";CVS(PRICE$)
70 PRINT CITY$: PRINT
80 GOTO 30
```

For further information on files, see Chapter 5.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

GOSUB...RETURN

Format: GOSUB <line number>
.
.
.
RETURN

Purpose: To branch to and return from a subroutine.

Remarks: Frequently, a program will need to execute a group of statements more than one time. A subroutine allows the group of statements to be keyed in only once and yet be accessed from different places in the program. The GOSUB statement is used to direct program flow to the subroutine.

<line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC/125 to branch back to the statement following the most recently executed GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO

Format: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
?Out of data in 10
Ok
```

IF...THEN[...ELSE]/IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: If <expression> GOTO <line number>

[ELSE <statement(s) > |<line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested, but nesting is limited by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT
     "LESS THEN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0) <1.0E-6 THEN...
```

rather than

```
IF A=1.0 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET#1,I
```

The above statement GETs record number I if I is not zero.

Example 2:

```
100 IF(I<20) * (I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

INPUT

Format: **INPUT [;][<"prompt string">;],<list of variables>**

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

If a single variable is requested, a <carriage return> may be entered to indicate the default values of 0 for numeric input and null for string input. Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?  5     (The 5 was typed in by the user
          in response to the question mark.)
 5 SQUARED IS 25
Ok
```

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS ";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946
OK
```

```
10 INPUT "ENTER THREE VALUES: ",A,B,C
20 D=(A+B+C)/3
30 PRINT "THE AVERAGE IS ";D
RUN
ENTER THREE VALUES:  5,10,9    (User types 5,10,9)
THE AVERAGE IS  8
```

```
10 INPUT; "ENTER EMPLOYEE NUMBER";A
20 IF A<25 THEN PRINT "INCORRECT VALUE"
.
.
.
RUN
ENTER EMPLOYEE NUMBER? 5INCORRECT VALUE (User types 5)
.
.
.
etc.
```

INPUT#

Format: INPUT #<file number>,<variable list>

Purpose: To read data items from a sequential disc file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC/125 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus a quoted string must not contain a quotation mark as a character. If the first character of a string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example:

```
10 OPEN "I",#1,"BUDG"
20 INPUT #1,CHCKNUM$,PAYEE$
30 PRINT CHCKNUM$;PAYEE$
40 GOTO 20
RUN
2134          ELECTRIC COMPANY
2568          GAS BILL
OK
```

For further information see Chapter 5.

KILL

Format: KILL <filename>

Purpose: To delete a file from disc.

Remarks: <filename> is a string expression. You must supply the CP/M file type .BAS. If <filename> is a literal then it must be enclosed in double quotation marks (").

If a KILL statement is given for a file that is currently OPEN, the file will be closed and then deleted.

KILL is used for all types of disc files: program files, random data files and sequential data files.

Example:

```
200 KILL "DATA1.BAS"
```

See also Chapter 5.

LET

Format: [LET] <variable>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Only the leftmost equal sign in an expression is used for assignment. Any additional equal signs in an expression are treated as relational operators. For example when the expression:

A=B=C

is evaluated, the value of A will be set to -1 (true) B is equal to C.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
      .
      .
      .
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
      .
      .
      .
```

LINE INPUT

Format: LINE INPUT[;][<"prompt string">;]<string variable>

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing <CTRL>-C. BASIC/125 will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example:

```
80 LINE INPUT "CUSTOMER INFORMATION?";C$
90 PRINT "YOU ENTERED ";C$
.
.
.
RUN
CUSTOMER INFORMATION? BOB PATTERSON     56988
YOU ENTERED  BOB PATTERSON     56988
```

LINE INPUT#

Format: LINE INPUT #<file number>,<string variable>

Purpose: To read an entire line (up to 254 character), without delimiters, from a sequential disc data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to, but not including, a carriage return. It then skips over the carriage return or carriage return/line feed sequence, and the next LINE INPUT# reads all following characters up to the next carriage return. (If a line feed/-carriage return sequence is encountered, it is preserved.) For example, if a file contains the following bytes:

A	CR	LF	B	CR	C	LF	D	CR	LF	E	LF	CR	F	CR	LF

then the following program:

```
10 OPEN "I",#1,"FILE"
20 FOR J=1 TO 4
30 LINE INPUT #1,C$
    .
    .
    .
200 NEXT J
```

will at line 30, read C$, in successive times, as:

```
1st time - A

2nd time - B

3rd time - C  LF  D

4th time - E  LF  CR  F
```

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC/125 program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "0",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES  234,4   MEMPHIS
LINDA JONES    234,4    MEMPHIS
Ok
```

LIST AND LLIST

Format: LIST [<line number>[-[<line number>]]]
or
LLIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the terminal, or using LLIST, at the line printer.

Remarks: BASIC/125 always returns to command level after a LIST or LLIST is executed.

1. If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing <CTRL>-C.)

2. If <line number> is included then only the specified line is listed.

3. If <line number>- is specified, then that line and all higher-numbered lines are listed.

4. If -<line number> is specified, then all lines from the beginning of the program through that line are listed.

5. If <line number>-<line number> is specified, then the entire range is listed.

(NOTE: LLIST assumes a 132-character wide printer.)

Examples:

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.
LIST 150-	Lists all lines from 150 to the end.
LIST -100	Lists all lines from the lowest number through 100.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

LOAD

Format: LOAD <filename>[,R]

Purpose: To load a file from disc into memory.

Remarks: <filename> is a string expression which is equal to the one used when the file was SAVEd. If <filename> is a literal then it must be enclosed in double quotation marks ("). The CP/M file type is automatically supplied if it is not included in <filename>.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disc data files.

Example:

```
LOAD "MYPROG",R
```

LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer.

LPRINT assumes a 132-character-wide printer.

LSET AND RSET

Format: LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKDS functions, in Chapter 4.

Examples:

```
10 OPEN "R",#1,"FILE",24
20 FIELD #1,4 AS AMT$,20 AS DESC$
30 INPUT "PRODUCT CODE";CODE%
40 INPUT "PRICE";PRICE
50 INPUT "DESCRIPTION";DSCRPN$
60 LSET AMT$=MKS$(PRICE)
70 LSET DESC$=DSCRPN$
80 PUT #1,CODE%
90 GOTO 30
```

See also Chapter 5.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACES(20)
120 RSET A$=NOTE$
```

right-justify the string NOTE$ in a 20-character field. This can be very handy for formatting printed output.

MERGE

Format: MERGE <filename>

Purpose: To merge a specified disc file into the program currently in memory.

Remarks: <filename> is a string expression which is equal to the name used when the file was SAVEd. If <filename> is a literal then it must be enclosed in double quotation marks. The CP/M file type of .BAS is automatically provided if it is not present in <filename>. The file must have been SAVEd in ASCII format. If not, a "Bad file mode" error occurs.

If any lines in the disc file have the same line numbers as lines in the program in memory, the lines from the file on disc will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disc into the program in memory.)

BASIC/125 always returns to command level after executing a MERGE command.

Example:

```
30 'THESE LINES WILL REPLACE THE OLD ONES   (FILE2)
40     VALUE=A*D/I
50     IF VALUE>300 THEN LIMIT=50

LOAD "FILE1"
LIST
10 'THIS IS FILE NUMBER ONE                 (FILE1)
20 FOR I=1 TO 10
30     'THESE LINES WILL BE REPLACED
40     H=K*J
60 NEXT I
70 PRINT "DONE"
OK
MERGE "FILE2"
OK
LIST
10 'THIS IS FILE NUMBER ONE
20 FOR I=1 TO 10
30 'THESE LINES WILL REPLACE THE OLD ONES   (NEW
40     VALUE=A*D/I                           FILE1)
50     IF VALUE>300 THEN LIMIT=50
60 NEXT I
70 PRINT "DONE"
OK
```

MID$

Format: MID$(<string expl>,n[,m])=<string exp2>

where n and m are integer expressions and <string expl> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. (See Chapter 4.)

NAME

Format: NAME <old filename> AS <new filename>

Purpose: To change the name of a disc file.

Remarks: <old filename> is a string expression which is equal to the name of the file when it was SAVEd or OPENed. <new filename> is also a string expression. If either <old filename> or <new filename> is a literal then they must be enclosed in double quotation marks ("). <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disc, in the same area of disc space, with the new name. (You must supply the CP/M file type of .BAS)

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In the above example, the file that was formerly named ACCTS will now be named LEDGER.

```
NAME "B:SALES" AS "B:STATS"
NAME "DEMO.BAS" AS "TEST.BAS"
```

NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC/125 always returns to command level after a NEW is executed.

NULL

Format: NULL <integer expression>

Purpose: To set the number of nulls to be printed at the end of each line. This applies to both the display and the printer.

Remarks: The default is 0. Some devices require that a specified number of null (blank) characters be printed at the end of each line. When using the Hewlett Packard periphals recommended for the HP 125, it is not necessary to use the NULL statement. For devices connected to serial port #2, the number of nulls can be set using the HP 125's configuration menu. The Number of nulls specified in the configuration are sent after every control character (i.e. for all ASCII codes which are greater than 00H and less than 20H, which includes carriage return), whereas the number of nulls sent by using the NULL statement are sent only at the end of a line.

Example:

```
NULL 2
```

ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors) , will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC/l25 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
.
.
.
1000 IF ERR=58 THEN PRINT "FILE ALREADY EXISTS"
1010 IF ERL=110 THEN RESUME 100
```

ON...GOSUB/ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example:

```
20 INPUT "ENTER SELECTION FROM MENU";A
30 ON A GOTO 50,70,90
40 PRINT "INVALID CHOICE": GOTO 20
50 PRINT "YOU CHOSE SELECTION NUMBER ONE"
.
.
.
70 PRINT "YOU CHOSE SELECTION NUMBER TWO"
.
.
.
```

OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Purpose: To allow access to a disc file (read and/or write).

Remarks: A disc file must be OPENed before any read or write operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first charcter is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disc I/O statements to the file.

<filename> is a string expression containing the name of the file, including the CP/M file type .xxx if appropriate, and the drive specifier if the file is not on the current disc. If <filename> is a literal then it must be enclosed in double quotation marks.

<reclen> is an integer expression which, if included, sets the record length for random files. The default maximum record length is 128 bytes. (NOTE: Maximum record length can also be set by using the /S option to the BASIC command.)

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

To add records to an existing sequential file which already contains data, you cannot simply OPEN the file in "O" mode and start PRINTing to it. As soon as a file is OPENed in "O" mode, it's current contents are destroyed. See Adding Data To A Sequential File, Chapter Five, for information on how to add data to an existing sequential file.

Example:

```
10 OPEN "I",2,"INVEN"
20 INPUT#2,PART$,DESC$
30 PRINT PART$;DESC$
40 GOTO 20
RUN
A2345G  WING NUT
B2399Z  35mm SCREW
.
.
.
```

See also Chapter 5

OPTION BASE

Format: OPTION BASE n where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

POKE

Format: POKE I,J
where I and J are integer expressions

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range 0 to 65535.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Chapter 4

POKE and PEEK are useful for effecient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
10 POKE &H5A00,&HFF
```

Places the value hex "FF" (binary all 1's for 8 bits) into absolute memory location hex "5A00".

PRINT

Format: PRINT [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Literals must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC/125 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC/125 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.

A question mark may be used in place of the word PRINT in a PRINT statement. When the program is listed back again, BASIC/125 will replace ? with PRINT.

Note: To send output to a printer use the LPRINT and LPRINT USING statements. (See also Chapter 1, Printing Operations.)

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0            -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

PRINT USING

Format: PRINT USING <string exp>;<list of expressions>

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers. The word PRINT may be replaced with ? when entering program lines. BASIC/l25 will replace it with PRINT when the program is LISTed.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the field is longer than the string, the string will be left-justified in the field and padding with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A&="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234,-3
10.20    5.30   66.79    0.23   -3.00
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##  ";-68.95,2.4,55.6,-.9
-68.95   +2.40  +55.60   -0.90

PRINT USING "##.##-  ";-68.95,22.449,-7.01
68.95-  22.45    7.01-
```

****** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#  ";12.39,-0.9,765.1
*12.4  *-0.9  765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number.

The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50
```

```
PRINT USING "####.##,";1234.5
1234.50,
```

^^^^ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02
```

```
PRINT USING ".####^^^^-";888888
.8889E+06
```

```
PRINT USING "+.##^^^^";123
+12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22
```

```
PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

PRINT# AND PRINT# USING

Format: PRINT#<file number>,[USING<string exp>;]<list of exps>

Purpose: To write data to a sequential disc file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described for the PRINT USING statement. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disc. An image of the data is written to the disc, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disc, so that it will be input correctly from the disc.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disc.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disc, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disc. Because there are no delimeters, this could not be INPUT# as two separate strings. To correct the problem, insert explicit delimeters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disc is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disc surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$="   93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disc:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" TO A$ AND "AUTOMATIC   93604-1" to B$. To separate these strings properly on the disc, write double quotes to the disc image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disc:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement

```
INPUT#1,A$;B$
```

would input "CAMERA, AUTOMATIC" to A$ and "   93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disc file. For example:

```
PRINT#1,USING"###.##,";J;K;L
```

For more examples using PRINT#, see Chapter Five.

See also WRITE#.

PUT

Format: PUT [#] <file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disc file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example:

```
10 OPEN "R" ,#1,"BDGT",30
20 FIELD #1,18 AS PAYEE$, 4 AS AMT$, 8 AS DATE$
30 INPUT "ENTER CHECK NUMBER";CK%
40 INPUT "PAYEE";PAY$
50 INPUT "DOLLAR AMOUNT";A
60 INPUT "DATE";D$
70 LSET PAYEE$=PAY$
80 LSET AMT$=MKS$(A)
90 LSET DATE$=D$
100 PUT #1,CK%
110 GOTO 30
```

For further information see Chapter Five.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement. In the case of WRITE#, BASIC/125 pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

RANDOMIZE

Format: RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC/125 suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Examples:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3 (user types 3)
 .88598   .484668    .586328    .119426    .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user types 4
for new sequence)
 .803506    .162462    .929364    .292443    .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same sequence
as first RUN)
 .88598   .484668    .586328    .119426    .709225
Ok
```

READ

Format: READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables. (See DATA).

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE).

Example 1:

```
 .
 .
 .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
 .
 .
 .
```

The above program segment READS the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08,A(2) will be 5.19, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY            STATE           ZIP
DENVER,         COLORADO        80211
```

The above program READs string and numeric data from the DATA statement in line 30.

REM

Format: REM <remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of REM.

Example:

```
     .
     .
     .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
     .
     .
     .
```

Or

```
120 'CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
```

Or

```
120 FOR I=1 TO 20    'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
     .
     .
     .
```

RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15, 30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET COMMAND

Format: RESET

Purpose: To allow you to access a disc which has just been inserted into a disc drive or to close all disc files and to write the directory information to a flexible disc before it is removed from the disc drive.

Remarks: It is mandatory that you execute a RESET command after inserting a new disc.

Always execute a RESET command before removing a disc from a disc drive. Otherwise, when the disc is used again it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every flexible disc with open files.

RESTORE

Format: RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example: The following program causes an "OUT OF DATA" error-

```
10 READ A,B,C
20 READ D,E,F
30 DATA 57,68,79
RUN
?Out of DATA in 20
OK
```

To reset the data pointer, and cause the data to be read again, add a RESTORE statement:

```
10 READ A,B,C
20 RESTORE 40
30 READ D,E,F
40 DATA 57,68,79
RUN
OK
```

RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
   .
   .
   .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY AGAIN"
1000 RESUME 80
   .
   .
   .
```

RUN

Format 1: RUN [<line number>]

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC/l25 always returns to command level after program execution is complete.

Example: RUN

Format 2: RUN <filename>[,R]

Purpose: To load a file from disc into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (The CP/M file type of .BAS is supplied for you.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example:

```
RUN "NEWFIL",R
```

For further information on files, see Chapter Five.

SAVE

Format: SAVE <filename>[,A \ ,P]

Purpose: To save a program file on disc.

Remarks: <filename> is a quoted string. The file is written to the current disc. To save a file on another disc include the CP/M drive specifier as part of <filename>. The default CP/M file type of .BAS is supplied for you. If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disc, but some disc access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as TYPE may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:

```
SAVE"COM2",A
SAVE"PROG",P
SAVE "B:BDGT"
```

For further information, see Chapter Five.

WHILE...WEND

Format:

```
WHILE <expression>
  .
  .
[<loop statements>]
  .
  .
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS=0
120         FOR I=1 TO J-1
130                 IF A(I)<=A(I+1) then 140
                    SWAP A(I),A(I+1) :FLIPS=1
140         NEXT I
150 WEND
```

WIDTH

Format: WIDTH [LPRINT] <integer expression>

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 1 to 255. The default width is 80 characters. When doing program development, set WIDTH to 255 so that program lines can automatically be extended over more than one physical line without a carriage return, or line feed. (See Chapter 1 for more information.)

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. The value returned by the POS or LPOS functions will be 0 anytime after the 255th character is printed on a line.

Example:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

WRITE

Format: WRITE[<list of expresions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they should be separated by commas.

When the printed lines are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

WRITE#

Format: WRITE#<file number>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disc and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disc.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disc:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

CHAPTER 4

BASIC/125 FUNCTIONS

The intrinsic functions provided by BASIC/125 are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC/125 will round the fractional portion and use the resulting integer.

NOTE: Only integer, string, and single precision results are returned by functions.

ABS

Format: ABS(X)

Action: Returns the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))
 35
Ok
```

ASC

Format: ASC(X$)

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix B for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

ATN

Format: ATN(X)

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

CDBL

Format: CDBL(X)

Action: Converts X to a double precision number.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

CHR$

Format: CHR$(I)

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix B.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message.

Example:

```
PRINT CHR$(66)
B
Ok
```

Or for example, the following escape sequence could be used to home the cursor and clear the display.

```
PRINT CHR$(27)+"H"+CHR$(27)+"J"
```

See the ASC function for ASCII-to-numeric conversion.

CINT

Format: CINT(X)

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

COS

Format: COS(X)

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

CSNG

Format: CSNG(X)

Action: Converts X to a single precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

CVI, CVS, CVD

Format: CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)

Action: Convert string values to numeric values. Numeric values that are read in from a random disc file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

See also MKI$, MKS$, MKD$, and Chapter 5.

EOF

Format: EOF(<file number>)

Action: Returns -1 (true) if the end of a file has been reached. Use EOF to test for end-of-file while INPUTing, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

ERR and ERL Variables

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
If ERR = error code THEN ...

If ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC/125's error codes are listed in Appendix A.

EXP

Format: EXP(X)

Action: Returns e (where e=2.71828...) to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

FIX

Format: FIX(X)

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:

```
PRINT FIX(58.75)
 58
Ok
```

```
PRINT FIX(-58.75)
-58
Ok
```

FRE

Format: FRE(0)
FRE(X$)

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC/125.

FRE("") forces the system to reorganize memory used by BASIC, so that no space is used by unreferenced variables, and then returns the number of free bytes. BE PATIENT: This process may take from 1 to 1-1/2 minutes. BASIC will not initiate this process until all free memory is used up. Therefore, using FRE("") periodically will result in shorter delays for each memory reorganization.

Example:

```
PRINT FRE(0)
 14542
Ok
```

HEX$

Format: HEX$(X)

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

INKEY$

Format: INKEY$

Action: Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through to the program except for Control-C, which terminates the program.

Example:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

INPUT$

Format: INPUT$(X[,[#]Y])

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:

```
  .
  .
  .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="S" THEN 700 ELSE 100
  .
  .
  .
```

INSTR

Format: INSTR([I,]X$,Y$)

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I to 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

NOTE: If I=0 is specified, the error message "ILLEGAL FUNCTION CALL" will be returned.

INT

Format: INT(X)

Action: Returns the largest integer <=X.

Examples:

```
PRINT INT(99.89)
 99
Ok
```

```
PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

LEFT$

Format: LEFT$(X$,I)

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC/125"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

LEN

Format: LEN(X$)

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:

```
20 PRINT LEN(X$)
30 X$ = "PORTLAND, OREGON"
40 PRINT LEN(X$)
RUN
0
16
OK
```

LOC

Format: LOC(<file number>)

Action: With random disc files, LOC returns the last record number used in a GET or PUT statement. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

LOF

Format: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example:

```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

LOG

Format: LOG(X)

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:

```
PRINT LOG(45/7)
 1.86075
Ok
```

LPOS

Format: LPOS(X)

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

MID$

Format: MID$(X$,I[,J])

Action: Returns a string of length J characters from X$ beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
 Ok
```

Also see the LEFT$ and RIGHT$ functions.

NOTE: If I=0, or I or J isn't in the range of 1 to 255, then the error message "ILLEGAL FUNCTION CALL" will be returned.

MKI$, MKS$, MKD$

Format: MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
90 AMT=(K+T)
100 FIELD #1, 4 AS D$, 20 AS N$
110 LSET D$ = MKS$ (AMT)
120 LSET N$ = A$
130 PUT #1
 .
 .
```

See also CVI, CVS, CVD, and Chapter 5.

OCT$

Format: OCT$(X)

Action: Returns a string which reprsents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

PEEK

Format: PEEK(I)

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65535. PEEK is the complementary function to the POKE statement.

Example:

```
A=PEEK(&H5A00)
```

POS

Format: POS(I)

Action: Returns the current cursor position. The leftmost position is 1. I is a dummy argument.

Example:

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function, and WIDTH statement.

RIGHT$

Format: RIGHT$(X$,I)

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:

```
10 A$="DISC BASIC/125"
20 PRINT RIGHT$(A$,9)
RUN
BASIC/125
Ok
```

Also see the MID$ and LEFT$ functions.

RND

Format: RND[(X)]

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Chapter 3). (However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

SGN

Format: SGN(X)

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example:

```
ON SGN(X)+2 GOTO 100,200,300 branches to 100 if
X is negative, 200 if X is 0 and 300 if X is
positive.
```

SIN

Format: SIN(X)

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision.
COS(X)=SIN(X+3.14159/2).

Example:

```
PRINT SIN(1.5)
 .997495
Ok
```

SPACE$

Format: SPACE$(X)

Action: Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:

```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

SPC

Format: SPC(I)

Action: Prints I blanks on the display. SPC may only be used with PRINT and LPRINT statements. If I<0 then SPC(I) prints a null string. SPC(I) for I>255 prints blanks (I mod 255). I is rounded to an integer to determine the number of blanks to print.

Example:

```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

Also see the SPACE$ function.

SQR

Format: SQR(X)

Action: Returns the square root of X. X must be >=0.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

STR$

Format: STR$(X)

Action: Returns a string representation of the value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

STRING$

Formats: STRING$(I,J)
STRING$(I,X$)

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$. I must be a positive integer in the range 0 through 255.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

TAB

Format: TAB(I)

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. I<1 is treated as I=1. I is first rounded, then I=I mod 256 is calculated. The The resulting I is used. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T. JONES","$25.00"
RUN
NAME                      AMOUNT

G.T. JONES                $25.00
Ok
```

TAN

Format: TAN(X)

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

USR

Format: USR[<digit>](X)

Action: Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
    .
    .
    .
```

VAL

Format: VAL(X$)

Action: Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and line-feeds from the argument string. For example,

VAL(" -3")

returns -3.

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
    PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL (ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
   .
   .
   .
```

See the STR$ function for numeric to string conversion.

VARPTR

Format 1: VARPTR(<variable name>)

Format 2: VARPTR(#<file number>)

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2: For sequential files, returns the starting address of the disc I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Example:

```
100 X=USR(VARPTR(Y))
```

CHAPTER 5

FILES

File procedures for the beginning BASIC/125 user are examined in this chapter. If you are new to BASIC/125 or if you're getting file related errors, read through these procedures and program examples to make sure you're using all the file statements correctly.

The CP/M operating system will append the default file type .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

Program File Commands

These are the commands and statements used in program file manipulation:

SAVE <filename>[,A]	Writes to disc the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD <filename>[,R]	Loads the program from disc into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

RUN <filename>[,R]	RUN <filename> loads the program from disc into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.
MERGE <filename>	Loads the program from disc into memory but does not delete the current contents of memory. The program line numbers on disc are merged with the line numbers in memory. If two lines have the same number, only the line from the disc program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disc. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename>	To change the name of a disc file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

Protected Files

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

Disc Filenames

Disc filenames follow the normal CP/M naming conventions. All filenames may include A: or B: as the first two characters to specify a disc drive, otherwise the currently selected drive is assumed. A default file type of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

Disc Data Files — Sequential & Random Access

There are two types of disc data files that may be created and accessed by a BASIC/125 program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
OPEN    PRINT#          INPUT#          WRITE#
        PRINT# USING    LINE INPUT#

CLOSE   EOF     LOC
```

The following program steps are required to create a sequential file and access the data in the file:

1.	OPEN the file in "O" mode.	`OPEN "O",#1,"DATA"`
2.	Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	`PRINT#1,A$;B$;C$`
3.	To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	`CLOSE #1` `OPEN "I",#1,"DATA"`
4.	Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

Program 5-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SHERLOCK HOLMES
DEPARMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? SUPER MANN
DEPPARTMENT? SHIPPING
DATE HIRED? 06/14/78
etc.
```

PROGRAM 5-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program 5-2. It accesses the file "DATA" that was created in Program 5-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM 5-2 - ACCESSING A SEQUENTIAL FILE

Program 5-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disc with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disc without explicit delimiters. The comma at the end of the format string serves to separate the items in the disc file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

Adding Data To A Sequential File -

If you have a sequential file residing on disc and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents.

The following procedures can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. Write the new information to "COPY".
5. CLOSE both files.
6. KILL "NAMES".
7. Rename "COPY" as "NAMES" and CLOSE it.
8. Now there is a file on disc called "NAMES" that includes all the previous data plus the new data you just added.

Program 5-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disc file. Remember, LINE INPUT# will read in characters from the disc until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
5  FLAG=1 'ASSUME FILE "NAMES" EXISTS
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN GOTO 110
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT #2,N$
170 PRINT #2,A$
180 PRINT #2,B$
190 PRINT : GOTO 120
200 CLOSE
202 IF FLAG=0 THEN 210 'SKIP KILL IF "NAMES" NEVER EXIST
205 KILL "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":FLAG=0
                RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM 5-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disc, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere in the file -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

Creating a Random File

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

   NOTE: The maximum logical record number is 32767. If a record is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed. (See /S option to BASIC command, Chapter 3.)

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD #1 20 AS N$, 4 AS A$,
         8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disc using the PUT statement.

```
PUT #1,CODE%
```

Look at Program 5-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE: Do not use a FIELDED STRING variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=0 THEN END
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM 5-4 - CREATE A RANDOM FILE

Access a Random File

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE#,32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1, 20 AS N$,
       4 AS A$, 8 AS P$
```

NOTE: In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

```
PRINT N$
PRINT CVS(A$)
```

Program 5-5 accesses the random file "FILE" that was created in Program 5-4. By inputting the two-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
45 IF CODE%=0 THEN END
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM 5-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number". The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 5-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
190 PRINT 7,"END PROGRAM"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>7) THEN PRINT
        "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680,890
240 GOTO 130
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$######.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%) <0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
        " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
715 IF ASC(F$)=255 THEN GOTO 730
720 IF CVI(Q$) <CVI(R$) THEN PRINT D$;" QUANTITY";
          CVI(Q$) TAB (50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
          GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

APPENDIX A

ERROR CODES AND ERROR MESSAGES

Code	Message
1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement.
2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.

5 Illegal function call
A parameter that is out of range is passed to a math or string function. An error 5 may also occur as the result of:

1. a negative or unreasonably large subscript
2. a negative or zero argument with LOG
3. a negative argument to SQR
4. a negative mantissa with a non-integer exponent
5. a call to a USR function for which the starting address has not yet been given
6. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

6 Overflow
The result of a calculation is too large to be represented in BASIC/125's number format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 Undefined line number
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

9 Subscript out of range
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate Definition
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of exponentiation results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

14 Out of string space
String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

15 String too long
An attempt is made to create a string more than 255 characters long.

16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,

2. has been modified during a break in execution, or

3. does not exist

18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in BASIC/125. Report to your HP 125 service office the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disc.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disc I/O error
An I/O error occurred on a disc I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disc.

61 Disc full
All disc storage space is in use.

62 Input past end
An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all directory entries are full.

APPENDIX B

REFERENCE TABLES

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	'	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	_
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Delete

HP 125 Escape Sequences

The following is a summary of the escape codes available on the HP 125 system. Refer to the System Reference Manual for a detailed explanation of the function of each sequence.

Two character escape sequences:

Sequence	Description
ESC 0	Print contents of display memory to selected printer.
ESC 1	Set tab at cursor column.
ESC 2	Clear tab.
ESC 3	Clear all tabs.
ESC 4	Set left margin.
ESC 5	Set right margin.
ESC 9	Clear all margins.
ESC @	Delay one second.
ESC A	Cursor Up.
ESC B	Cursor Down.
ESC C	Cursor Right.
ESC D	Cursor Left.
ESC E	Hard reset TPU.
ESC F	Home Down.
ESC G	Return.
ESC H	Home Up cursor.
ESC I	Tab.
ESC J	Clear Display.
ESC K	Clear Line.
ESC L	Insert line.
ESC M	Delete line.
ESC P	Delete character.
ESC Q	Turn on insert character mode.
ESC R	Turn off insert character mode.
ESC S	Roll Up Display.
ESC T	Roll Down Display.
ESC U	Next Page.
ESC V	Previous Page.
ESC Y	Display Functions On.
ESC Z	Display Functions Off.
ESC ^	Send Primary Terminal Status.
ESC `	Relative Cursor Sense.

Two character escape sequences (continued)

```
    ESC a           Absolute Cursor Sense.
    ESC b           Enable Keyboard.
    ESC c           Disable Keyboard.
    ESC d           Enter Line.
    ESC f           Modem Disconnect.
    ESC g           Soft Reset TPU.
    ESC h           Home Up Cursor.
    ESC i           Back Tab Cursor.
    ESC j           Display User Definition Keys Menu.
    ESC k           Exit User Keys Definition Menu.
    ESC x           Perform datacomm self-test.
    ESC z           Perform TPU self-test.
```

Multi-character escape sequences:

```
    ESC & a         Cursor Addressing.
                <col number> c  -> Set Column
                <row number> r  -> Set Absolute Row
                <row number> y  -> Set Relative Row
    ESC & d         Display Enhancement.
                @               -> Clear Enhancement
                <A|B|C...|O>    -> Set Enhancement
    ESC & f         Define User Softkeys.
                < 0 | 1 | 2> a  -> Set Key Attribute
                < length   > d  -> Label Length
                <1|2|...|8 > E  -> Execute Specified Key
                <1|2|...|8 > k  -> Select Key to be Defined
                < length   > l  -> Definition String
                <label string>  -> Label Display Definition
                <defin string>  -> Softkey Execution Definition
    ESC & i         Device Assignment.
                <bufr numbr> d  -> Destination
                <bufr numbr> s  -> Source
                <mode numbr> m  -> Operation Mode

    ESC & j         Softkey Display Control.
                        @  -> Turn Off Softkey Labels
                        A  -> Turn On Softkey Labels
                        B  -> Turn On User Softkey Labels
                        R  -> Unlock Softkey Label Set
                        S  -> Lock Softkey Label Set
```

Multi-character escape sequences (continued)

ESC & k Set Terminal Straps.
All parameters 0 or 1. 0 -> Disable strap
1 -> Enable Strap

a -> Auto Line Feed
c -> Caps Lock Mode
d -> Bell
j -> Screen refresh frequency
l -> Local Echo
m -> Modify All Mode
n -> Space overwrite latch
p -> Caps Mode
q -> Key Click
r -> Remote Mode

ESC & p Peripheral Device Control.
^ -> Status Request
b -> Print One Line
f -> Print Page
m -> Print Memory
<control num> c -> Data Log Function
<control num>=11 -> Log Bottom
=12 -> Log Top
=13 -> Turn Logging Off
=17 -> Enable Report Format
=18 -> Enable Metric Print
=19 -> Disable Report/Metric Format

ESC & q Terminal Configuration Control.

< 0 | 1 > 1 -> Configuration Lock

ESC & s Terminal Strap Control.
< 0 | 1 > a -> Transmit Function Key Sequences
< 0 | 1 > b -> Space Overwrite Enable
< 0 | 1 > c -> End of Line Cursor Wraparound
< 0 | 1 > g -> Reset G Handshake
< 0 | 1 > h -> Reset H Handshake
< 0 | 1 > l -> Self-test Inhibit

ESC * s ^ Terminal Identification

BASIC/125 Reserved Words

ABS
ASC
ATN
AUTO
CALL
CDBL
CHAIN
CHR$
CINT
CLEAR
CLOSE
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DELETE
DIM
ELSE
ELSE$
END
EOF
ERASE
ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
INKEY$
INP
INPUT
INPUT$
INSTR
INT
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LPRINT
LSET
MBASIC
MERGE
MID$
MKD$
MKI$
MKS$
NAME
NEW
NEXT
NULL
OCT$
ON
OPEN
OPTION
OUT
PEEK
POKE
POS
PRINT
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SPACE$
SPC
SQR
STEP
STRING$
SWAP
SYSTEM
TAB
TAN
THEN
TROFF
TRON
USING
USR
VAL
VARPTR
WAIT
WEND
WIDTH
WHILE
WRITE

APPENDIX C

ASSEMBLE LANGUAGE SUBROUTINES

BASIC/125 has the capability for interfacing with assembly language subroutines. The USR function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

Memory Allocation

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization of BASIC/125, using the BASIC command, use the /M option to enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). Or use the CLEAR command. (See Chapter 3 for more information on BASIC and CLEAR.) BASIC uses all memory available from its starting location up, so only the memory loactions above BASIC can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

USR Function Calls

The format of the USR function is

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, it is contained in the Floating Point Accumulator (FAC). The [H,L] register pair points to FAC-3.

If the argument is an integer:

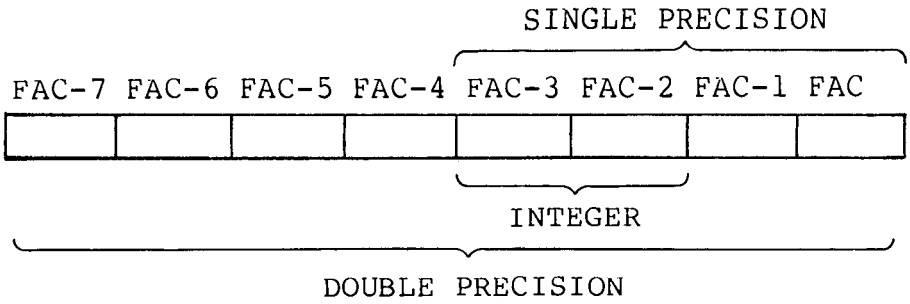
FAC-3 contains the lower 8 bits of the argument and FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and FAC-2 contains the middle 8 bits of mantissa and FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative). FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa (the implied leading 1).

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).



Below are examples of the number 9 expressed as an integer, single precision number, and double precision number. (Each byte is represented by 2 hexadecimal digits.)

	FAC-7				FAC-3			FAC
Integer:	?	?	?	?	09	00	?	?
Single:	?	?	?	?	00	00	10	84
Double:	00	00	00	00	00	00	10	84

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, concatenate a single null character (+"") to the string literal in the program. Example:

```
A$ = "BASIC/125"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
        PUSH    H       ;save value to be returned
        LHLD    105H    ;get address of MAKINT routine
        XTHL            ;save return on stack and
                        ;get back [H,L].
        RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
        LXI     H,SUB1  ;get address of subroutine
                        ;continuation
        PUSH    H       ;place on stack
        LHLD    103H    ;get address of FRCINT
        PCHL
SUB1: . . . . .
```

NOTE: The address of MAKINT is 105 hex and the address of FRCINT is 103 hex.

CALL Statement

BASIC/l25 user function calls may also be made with the CALL statement.

CALL <variable name>[(<argument list>)]

For more information see CALL, Chapter Three.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET". (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

This is similar, but not identical, to USR function calls. The arguments are NOT stored in the Floating Point Accumulator (FAC). For USR function calls, the pointer points to the low byte of the argument, like the CALL statement, for integers and single precision numbers. For double precision numbers, with the CALL statement, the pointer points to the low byte of the argument, while with the USR function, the pointer points to the "middle" of the argument. The pointers for both the USR function call and the CALL statement point to the low byte of a string descriptor (described above) for string arguments.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter 1 in HL.

    2. Parameter 2 in DE.

    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, then the code below may be helpful.

```
N       EQU     10      ;# OF PARAMETERS IN THIS EXAMPLE IS 10
SUBR    SHLD    P1      ;SAVE PARAMETER 1
        XCHG
        SHLD    P2      ;SAVE PARAMETER 2
        MVI     A,N-2   ;NO. OF PARAMETERS LEFT
        LXI     D,P3    ;POINTER TO LOCAL AREA
        MOV     H,B
        MOV     L,C     ;[H,L] - PTR TO PARAMS
AT1     MOV     C,M
        INX     H
        MOV     B,M
        INX     H       ;[B,C] - PARAM ADR
        XCHG            ;[H,L] POINTS TO LOCAL STORAGE
        MOV     M,C
        INX     H
        MOV     M,B
        INX     H       ;STORE PARAM IN LOCAL AREA
        XCHG            ;SINCE GOING BACK TO AT1
        DCR     A       ;TRANSFERRED ALL PARAMS?
        JNZ     AT1     ;NO, COPY MORE
        .
        [Body of Subroutine]
        .
        RET             ;RETURN TO CALLER
P1      DS      2       ;SPACE FOR PARAMETER 1
P2      DS      2       ;SPACE FOR PARAMETER 2
P3      DS      2*N-4   ;SPACE FOR PARAMETERS N-4
```

When accessing parameters in a subroutine, don't forget that they are POINTERS to the actual arguments passed.

NOTE

It is entirely up to the programmer to see that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines as well as those written in assembly language.

The listing below is of 2 Assembly language subroutines which demonstrate access from BASIC/125:

```
;Below are two example assembly language sub-
;routines, both for use by BASIC/125. The first
;example is a USR function call and the other
;is a CALL statement.
;
;In this example, the USR0 function entry point
;is at B200H, and an additional parameter is at
;B203H.  The normal parameter to be passeed via
;the USR0 call, is a string to be passed to an
;HPIB device.  This HPIB device might typically
;be a plotter.  The additional parameter at
;B203H is the HPIB device address.  For a
;plotter, for example, this typically would be
;address 5.  A calling sequence from BASIC
;might  be:
;
;550     POKE &HB203,5
;560     X=USR0(A$)
;
;where A$ had previously been set to the string
;desired to be output to the plotter.  This
;function returns a type integer return code:
;
;       0=All OK
;       1=HPIB address not between 1 and 31, in
;       2=HPIB device didn't accept data in
;         allotted time
;       3=Argument not of type "string"
;
;The CALL statement subroutine is one that will
;display "error" or "informational" messages in
;the message window (temporarily replacing the
;softkeys).  This call has 2 parameters, the
;first being the indicator:
;
;       0="error" message
;       1="informational" message
;
;(It must be an integer.)  The second parameter
;is the actual message, a string variable,
;padded at the front with 2 dummy characters
;(that are replaced with blanks by this sub-
;routine), that are not displayed.  No type
;checking is done on the parameters. A typical
;calling sequence from BASIC/125 might be:

;290     DEFINT E,I
;300     ERRORTYP=0
;310     INFOTYP=1
;320     PMSG=&HB204
```

```
                    ;330    MSG$="  ***THIS IS AN ERROR MESSAGE."
                    ;340    CALL PMSG(ERRORTYP,MSG$)
                    ;350    MSG$="  ***THIS IS AN INFO MESSAGE."
                    ;360    CALL PMSG(INFOTYP,MSG$)
                    ;
B200                        ORG     0B200H
                    ;
0005 =      BDOS            EQU     5           ;ENTRY POINT FOR SYSTEM
                                                ;FUNCTION CALLS
76FF =      HPBFN           EQU     76FFH       ;HPIB OUTPUT FUNCTION NUMBER
0003 =      STRTYP          EQU     3           ;ARGUMENT TYPE "STRING" IS 3
0105 =      MAKINT          EQU     105H        ;ADDRESS OF MAKINT ROUTINE
7BFF =      MSGNO           EQU     7BFFH       ;FUNCTION NUMBER FOR MESSAGES
                    ;
                    ;
            USR0:
B200 C307B2                 JMP     HPIB
          HPBAD:
B203                        DS      1
          CALL1:
B204 C366B2                 JMP     MSG
                    ;
                    ;
           HPIB:
B207 210000                 LXI     H,0
B20A 39                     DAD     SP          ;PUT STACK POINTER INTO HL
B20B 31C9B2                 LXI     SP,STCK     ;LOAD OUR OWN STACK POINTER
B20E E5                     PUSH    H           ;SAVE OLD STACK POINTER ON
                                                  NEW
                                                ;STACK
B20F FE03                   CPI     STRTYP      ;IS A = TYPE STRING?
B211 C254B2                 JNZ     HPBERR      ;IF NOT, GOTO HPIB ERROR
B214 3A03B2                 LDA     HPBAD       ;CHECK FOR VALID HPIB
                                                  ADDRESS
B217 FE20                   CPI     32          ;32=MAX VALID ADDRESS
B219 D260B2                 JNC     HPBER3      ;IF > 31, IT'S AN ERROR
B21C B7                     ORA     A           ;IF =0 (MAIN DISC DRIVE),
                                                ;IT'S AN ERROR
B21D CA60B2                 JZ      HPBER3
B220 IA                     LDAX    D           ;GET 1ST BYTE OF 3 BYTE
                                                  STRING
                                                ;DESCRIPTOR, THE LENGTH
B221 47                     MOV     B,A         ;AND SAVE IT IN B.
B222 13                     INX     D           ;POINT DE TO THE STRING
                                                ;POINTER
B223 EB                     XCHG                ;PUT DE INTO HL
B224 5E                     MOV     E,M         ;GET LOW BYTE ADDRESS
                                                  POINTER TO
                                                ;STRING
B225 23                     INX     H           ;POINT TO HI BYTE
B226 56                     MOV     D,M         ;GET HI BYTE OF ADDRESS
                                                  POINTER
                                                ;TO STRING
B227 EB                     XCHG                ;POINTER INTO HL
B228 B7                     ORA     A           ;IS LENGTH OF STRING =0?
```

```
B229 CA38B2          JZ     DONHPB    ;IF SO, GOTO DONE WITH HPIB
                                      ;(DONHPB)
          PLOOP:
B22C 5E              MOV    E,M       ;GET NEXT CHAR IN STRING
                                        AND PUT
                                      ;IT IN THE E REGISTER WHERE
                                        THE
                                      ;HP-ADD SYSTEM FUNCTION
                                        EXPECTS THE
B22D CD44B2          CALL   HPBCHR    ;SEND THIS CHAR TO HPIB
                                        DEVICE
B230 C25AB2          JNZ    HPBER2    ;IF ERROR WHILE OUTPUTTING
                                      ;GOTO HPBER2
B233 23              INX    H         ;POINT TO NEXT CHAR IN
                                        STRING
B234 05              DCR    B         ;DECREMENT COUNTER
B235 C22CB2          JNZ    PLOOP     ;IF NOT DONE, LOOP TO GET
                                        NEXT
                                      ;CHAR
          DONHPB:
B238 210000          LXI    H,0       ;LOAD HL WITH RETURN VALUE
                                        FOR OK
          USRRET:
B23B EB              XCHG             ;SAVE RETURN VALUE IN DE
B23C E1              POP    H         ;RETRIEVE PREVIOUSLY SAVED
                                        OLD
                                      ;STACK POINTER
B23D F9              SPHL             ;PUT OLD STACK POINTER INTO
                                      ;STACK POINTER REGISTER
B23E D5              PUSH   D         ;SAVE RETURN VALUE
B23F 2A0501          LHLD   MAKINT    ;GET ADDRESS OF MAKINT
                                        ROUTINE
B242 E3              XTHL             ;PUT RETURN VALUE BACK INTO
                                        HL
                                      ;AND PUT ADDRESS OF MAKINT
                                        ON
                                      ;TOP OF STACK
243 C9               RET              ;GOTO MAKINT, LET MAKINT
                                        RETURN
                                      ;INTEGER IN HL TO BASIC,
                                        AND
                                      ;LET MAKINT RETURN PROGRAM
                                      ;CONTROL TO BASIC
        HPBCHR:
B244 E5              PUSH   H         ;SAVE REGISTERS
B245 C5              PUSH   B
B246 01FF76          LXI    B,HPBFN   ;LOAD BC WITH FUNCTION
                                        NUMBER
                                      ;TO OUTPUT CHAR
B249 3A03B2          LDA    HPBAD
B24C 57              MOV    D,A       ;LOAD D WITH HPIB ADDRESS
B24D CD0500          CALL   BDOS      ;OUTPUT ONE CHAR BY MAKING
                                        SYSTEM
                                      ;CALL
B250 C1              POP    B         ;RESTORE REGISTERS
```

```
B251 E1              POP   H
B252 B7              ORA   A          ;WAS THERE AN ERROR WHILE
                                      ;OUTPUTTING TO THE HPIB?
B253 C9              RET   RET        ;RETURN WITH ERROR
                                        INDICATOR
                                      ;
          HPBERR:
B254 210100           LXI   H,1       ;RETURN ERROR CODE=1 MEANS
                                      ;ARGUMENT NO OF TYPE STRING
B257 C33BB2           JMP   USRRET
          HPBER2:
B25A 210200           LXI   H,2       ;RETURN ERROR CODE=2 MEANS
                                        HPIB
                                      ;DEVICE DIDN'T ACCEPT BYTES
                                        IN
                                      ;ALLOTTED TIME
B25D C33BB2           JMP   USRRET
          HPBER3:
B260 210300           LXI   H,3       ;RETURN ERROR CODE=3 MEANS
                                        BAD
                                      ;HPIB ADDRESS
B263 C33BB2           JMP   USRRET
          ;
          ;
          MSG:
B266 46              MOV   B,M        ;B=MSG TYP, O=ERR, 1=INFO
B267 210000          LXI   H,0        ;SAVE BASIC'S STACK
B26A 39              DAD   SP
B26B 22C7B2          SHLD  STCK-2     ;SAVE OLD STACK POINTER ON
                                        TOP
                                      ;OF NEW STACK
B26E 31C7B2          LXI   SP,STCK-2;LOAD NEW STACK POINTER
B271 EB              XCHG             ;HL=>STRING DESCRIPTOR
B272 7E              MOV   A,M        ;A=LEN OF STRING
B273 23              INX   H          ;POINT HL TO ADDRESS OF
                                        STRING
B274 5E              MOV   E,M
B275 23              INX   H
B276 56              MOV   D,M        ;DE=>STRING
B277 ED              DCR   A          ;LEN OF STRING IS = LEN OF
                                        MSG
                                      ;PLUS 2 PADDED CHARACTERS.
                                        LEN
                                      ;TO PASS TO FUNCTION IS LEN
                                        OF
                                      ;MSG PLUS 1 FOR THE
                                        INDICATOR,
                                      ;SO WE -1 FROM LEN OF
                                        STRING
B278 12              STAX   D         ;FILL FIRST BYTE OF STRING
                                        WITH
                                      ;LEN
B279 D5              PUSH   D         ;SAVE POINTER TO STRING
B27A 13              INX    D         ;POINT TO SECOND BYTE OF
                                        STRING
```

```
B27B 78              MOV    A,B
B27C 12              STAX   D          ;PUT INDICATOR INTO BUFFER
B27D 1B              DCX    D          ;POINT DE BACK TO FIRST
                                          BYTE OF
                                       ;BUFFER
B27E 01FF7B          LXI    B,MSGNO    ;LOAD FUNCTION NUMBER
B281 CD0500          CALL   BDOS       ;ISSUE FUNCTION CALL
B284 D1              POP    D          ;RESTORE POINTER TO STRING
B285 3E20            MVI    A,' '      ;REPLACE LEN AND INDICATOR
                                       ;WITH BLANKS
B287 12              STAX   D
B288 13              INX    D
B289 12              STAX   D
B28A E1              POP    H          ;HL=OLD STACK POINTER
B28B F9              SPHL              ;RESTORE OLD STACK POINTER
B28C C9              RET
          ;
B28D      STCK:                        ;60 BYTES ABOVE ARE STACK
                                          AREA
          ;
B2C9                 END
```

The following BASIC/125 program is an example of how to access an assembly langauge subroutine from a BASIC program. It uses the above routines to write to the plotter, and issue "error" and "informational" messages.

```
LIST
10 DEFINT E,I
20 BASE=&HB200
30 DEF USR0=BASE
40 PMSG=BASE+4
50 ERRORTYP=0
60 INFOTYP=1
65 REM Below sets HPIB address where USR0 will send data
70 POKE BASE+3,5
80 INPUT "Please type characters to send to the plotter.  ",A$
90 IF A$="" THEN END
100 X=USR0(A$)
110 IF X=0 THEN GOTO 80
120 IF X=1 THEN GOTO 170
130 IF X=2 THEN GOTO 200
140 MSG$=" ***Error in USR0 statement, argument type must be
     string."
150 CALL PMSG(ERRORTYP,MSG$)
160 STOP
170 MSG$="  ***Error in USR0 routine, HPIB address is
     invalid."
180 CALL PMSG(ERRORTYP,MSG$)
190 STOP
200 MSG$= ***Error in USR0 routine, HPIB device didn't accept
     data."
210 CALL PMSG(INFOTYP,MSG$)
220 GOTO 80
```

Included on the disc which contains BASIC/125 is a program called MEMSIZE. It will provide you with the correct parameter to use with the /M option to the BASIC command. To use it, simply type MEMSIZE and the length of your assembly routine (in decimal in hexadecimal). The program will then display the BASIC command and the highest memory location you should use. Examples

```
B>MEMSIZE                          (No size specified by user)

Use the following command:         (System displays this)

       BASIC /M:54271
or
       BASIC /M:&HD3FF
```

```
B>MEMSIZE 123

Use the following command:

       BASIC /M:54148

or

       BASIC /M:&HD384
```

```
B>MEMSIZE &h2DF

Use the following command:

       BASIC /M:53536

or

       BASIC /M:&HD120
```

APPENDIX D

MATHEMATICAL FUNCTIONS

Derived Functions

Functions that are not intrinsic to BASIC/125 may be calculated as follows.

Function	BASIC/125 Equivalent
SCANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(1/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE CONTANGENT	ARCCOT(X)=-ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=-EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1

INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E

CONVERTING PROGRAMS TO BASIC/125

If you have programs written in a BASIC other than BASIC/125, some minor adjustments may be necessary before running them with BASIC/125. Here are some specific things to look for when converting BASIC programs.

String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC/125 statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC/125 string concatenation.

In BASIC/125, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	BASIC/125
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	BASIC/125
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC/125 would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With BASIC/125, be sure all statements on a line are separated by a colon (:).

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

New Features In BASIC-80, Release 5.0

Although BASIC/125 is compatible with BASIC-80, the execution of BASIC/125 programs written under Microsoft BASIC-80, release 4.51 and earlier may be affected by some of the new features in release 5.0. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION, BASE, RANDOMIZE.

2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB (4.5) goes to the 5th position, A(1.5) yields A(2), and X=11.5 MOD 4 yields 0 for X).

3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

4. Division by zero and overflow no longer produce fatal errors.

5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used.

6. The rules for PRINTing single precision and double precision numbers have been changed.

7. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space.

8. Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.), or with a carriage return causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

9. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

10. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer.

11. The at-sign and underscore are no longer used as editing characters.

12. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of the line to be truncated if the line length is close to 255 characters.

13. BASIC programs may be saved in a protected binary format.

APPENDIX F

QUICK REFERENCE

BASIC/125 Commands and Statements — Syntax Reference

AUTO [<line number>[,<increment>]]

BASIC
or
BASIC [<filename>][/F:<number of files>][/M:<highest memory location>][/S:<maximum record size>]

CALL <variable name>[(<argument list>)]

CHAIN [MERGE]<filename>[,[<line number exp>][,ALL][,DELETE<range>]]

CLEAR [[,<expression1>][,<expression2>]]

CLOSE [#]<file number>[,[<file number...>]]

COMMON <list of variables>

CONT

DATA <list of constants>

DEF FN <name>[(<parameter list>)]=<function definition>

DEF <type> <range(s) of letters> where <type> is INT, SNG, DBL, or STR

DEF USR [<digit>]=<integer expression>

DELETE [<line number>][-<line number>]

DIM <list of subscripted variables>

END

ERASE <list of variables>

ERROR <integer expression>

FIELD [#]<file number>,<field width> AS <string variable>[,]

FILES [<filename>]

FOR <variable>=x to y [STEP z]
    .
    .
    .
NEXT [<variable>][,<variable>] where x, y, and z are numeric expressions.

GET [#]<file number>[,<record number>]

GOSUB <line number>
    .
    .
    .
RETURN

GOTO <line number>

IF <expression> THEN <statment(s)> | <line number>
    [ELSE <statement(s)> | <line number>]

IF <expression> GOTO <line number>
    [ELSE <statement(s)> | <line number>]

INPUT [;][<"prompt string">;]<list of variables>

INPUT# <file number>,<variable list>

KILL <filename>

[LET] <variable>=<expression>

LINE INPUT [;][<"prompt string">;]<string variable>

LINE INPUT# <file number>,<string variable>

LIST [<line number>[-[<line number>]]]
    or
LLIST [<line number>[-[<line number>]]]

LOAD <filename>[,R]

LPRINT [<list of expression>]

LPRINT USING <string exp>;<list of expressions>

LSET <string variable>=<string expression>

MERGE <filename>

MID$ (<string expl>,n[,m])=<string exp2>

NAME <old filename> AS <new filename>

NEW

NULL <integer expression>

ON ERROR GOTO <line number>

ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

OPTION BASE n where n is 1 or 0

POKE I,J where I and J are integer expressions

PRINT [<list of expressions>]

PRINT USING <string exp>;<list of expressions>

PRINT# <file number>,[USING<string exp>;]<list of exp>

PUT [#] <filenumber>[,<record number>]

RANDOMIIZE [<expression>]

READ <list of variables>

REM <remark>
or
' <remark>

RENUM [[<new number>][,[<old number>][,<increment>]]]

RESET

RESTORE [<line number>]

```
RESUME
    or
RESUME 0
    or
RESUME NEXT
    or
RESUME <line number>

RSET <string variable>=<string expression>

RUN [<line number>]
    or
RUN <filename>[,R]

SAVE <filename>[,A\,P]

STOP

SWAP <variable>,<variable>

SYSTEM

TRON

TROFF

WHILE <expression>
    .
    .
    .
    [<loop statements>]
    .
WEND

WIDTH [LPRINT] <integer expression>

WRITE <list of expressions>

WRITE# <file number>,<list of expressions>
```

BASIC/125 Functions — Syntax Reference

ABS (X)

ASC (X$)

ATN(X)

CDBL (X)

CHR$ (I)

CINT (X)

COS (X)

CSNG (X)

CVD (<8-byte string>)

CVI (<2-byte string>)

CVS (<4-byte string>)

EOF (<file number>)

ERL

ERR

EXP (X)

FIX (X)

FRE (0)

FRE (X$)

HEX$ (X)

INKEY$

INPUT$ (X[,[#]Y])

INSTR ([I,]X$,Y$)

INT (X)

**LEFT$** (X$,I)

LEN (X$)

LOC (<file number>)

LOF (<file number>)

LOG (X)

LPOS (X)

**MID$** (X$,I[,J])

MKD$(<double precision expression>)

MKI$ (<integer expression>)

MKS$(<single precision expression>)

OCT$ (X)

PEEK (I)

POS (I)

**RIGHT$** (X$,I)

RND[(X)]

SGN (X)

SIN (X)

SPACE$ (X)

SPC (I)

SQR (X)

STR$ (X)

**STRING$** (I,J) or STRING$(I,X$)

TAB (I)

TAN(X)

USR [<digit>](X)

VAL (X$)

VARPTR (<variable name>)
or
VARPTR (#<file number>)

APPENDIX G

INSTALLING BASIC/125

To install BASIC/125 on the HP 125 you need to perform the following steps:

1. Place the system disc into drive A (the left drive), and press the LOAD OP SYS function key to load the CP/M operating system. The Welcome screen will appear and display the application menu. If the message Welcome? appears on the screen, or if you need more detailed information on how to load the operating system, refer to the HP 125 Getting Started manual.

2. To place the Welcome program into installation mode, press the <SHIFT>, <CTRL>, and "@" keys simultaneously.

3. When the instruction screen appears, insert the BASIC/125 disc into drive B, and press the INSTALL APPL function key.

4. The message "Application Installation in Progress" will be displayed. It will take a few moments for BASIC/125 to be installed.

5. Once BASIC/125 has been installed, press the EXIT softkey and the new version of the Welcome program will be saved automatically on our system disc.

6. When installation is complete, the Welcome screen will appear and a function key label for BASIC/125 will be included on the application menu.

APPENDIX H

USING HP125 TERMINAL FEATURES IN BASIC

The terminal portion of the HP 125 can be 'programmed' to perform many of the functions of an intelligent terminal. By using these features, the BASIC programmer can be more productive by allowing the HP 125 to perform tasks which would otherwise need to be done within each application program.

You are probably aware of the various features available to the HP 125 user via the keyboard. Most of the features which can be performed at the keyboard can also be done under program control by using the 'escape sequences' described in Appendix C and below. Additionally, to offer compatibility with other Hewlett Packard terminal products, the escape sequences used by the HP 125 are the same sequences used on other Hewlett Packard display terminal products.

An escape sequence is simply a series of ASCII characters preceded by an 'escape' character, which has an ASCII code of 27. For example, printing an 'ESC h' to the HP 125 display will cause the cursor to move to the 'home' position at the upper left hand corner of the screen. Note that the 'escape' character will be represented throughout this appendix as 'ESC'.

Escape sequences can be classified into two categories: two character sequences, and multiple character sequences. In two character sequences, the entire escape sequence must match the sequence described in this manual. Multiple character sequences generally allow various parameters to be interchanged.

Multiple character sequences begin with an 'ESC' followed by a n ampersand character (&), ASCII 38. This sequence is indicated in this appendix as 'ESC &'. The characters that follow consist of one or more numeric character string parameters, each followed by an alphabetic character. All of these alpha characters should be lowercase, except the last alpha terminator in a given sequence. This is because the multiple character sequence parameter list can be combined or shortened, and the terminal will not execute any of the sequences until an uppercase alpha character is received.

An example is the cursor positioning escape sequence. One way of positioning the cursor in the upper left hand corner of the display is to use the 'home-up' sequence illustrated above. A second way is to use the 'cursor addressing' sequence to move the cursor to row zero, column zero. The sequence to perform this task is:

```
ESC & a 0 r 0 C
```

The upper case 'C' ends the sequence. To position the cursor to the top line, without affecting the column, you could send:

```
ESC & a 0 R
```

Notice that the 'C', or 'column', parameter was simply omitted: the upper case 'R' terminated the sequence.

One more thing about multiple character escape sequences: none require a fixed order of parameters. For example, the first escape sequence above is functionally identical to:

```
ESC & a 0 c 0 R
```

Which parameter comes first is not critical: that the last parameter ends with an uppercase letter is important.

Escape sequences will be illustrated with a blank between each of the characters. This is done for clarity, and the spaces should <u>not</u> be inserted in actual sequences.

There are one or two things to watch for when using escape sequences with BASIC/l25. First, remember that the 'PRINT' statement will force a 'carriage-return/line-feed' after every PRINT statement unless the string to be printed is followed by a semi-colon (;). If you print a sequence which positions the cursor, and forget to end the PRINT with a semi-colon, the cursor will automatically move to the next line.

Also, BASIC/l25 keeps track of the number of characters printed on each line so that a 'carriage return/line feed' can be added after every 80 characters. If you have been PRINTing escape sequences, you probably do not want these characters automatically added. By specifying the WIDTH command with a value of 255, BASIC will stop inserting the automatic '<CR><LF>' and permit your program to fully utilize terminal control sequences.

Sample Functions

An example of using escape sequences within a BASIC/125 program is illustrated below. This example uses only four of the possible sequences included in the table which follows. However, by using these sample functions as a model, you should be able to program any of the other functions available.

The function definitions have been entered on multiple lines just as you see them. If the program line is entered normally, each will use 80 characters and formatting the program listing as you see here is often impossible. However, by using a Control-J at the end of each 'logical' line, BASIC allows single line statements to be entered on more than one 'physical' line.

```
1000 'DEFINE ESCAPE SEQUENCES AS FUNCTIONS
1010 ESC$=CHR$(27)
1020 DEF FNHOME$=ESC$+"h"+ESC$+"J"
1030 DEF FNCURSOR$(C,R)=ESC$+&a"+STR$(C)+"c"+STR$(R)+"R"
1040 FNKEY$(K,A$,B$)=ESC$+"&f2a"+STR$(K)+"k"+STR$
        (LEN(A$))+"d"+STR$(LEN(B$))+"L"+A$+B$
1050 DEF FNIV$(A$)=ESC$+"&dB"+A$+ESC$+"&d@"
```

Before exploring how these functions might be used within a program, take a closer look at each one.

FNHOME$, when printed, executes a 'Home Up, Clear Display' sequence. This positions the cursor at the top of display memory, and clears the screen.

FNCURSOR$, when printed, positions the cursor to the row and column specified by R and C. Note that the string function 'STR$' must be used to convert the numeric values of C and R into a string representation of the desired values.

FNKEY$, when printed, allows any of the [USER KEYS] to be defined. The key to be defined is specified as K, the label is A$, and the definition is B$. Note that the string representation of the length of each field must be specified. As with the cursor function above, a numeric value must be converted to a string.

FNIV$, when printed, causes the string of characters in A$ to be printed in inverse video at the current cursor position. FNIV$ also verifies that only A$ will be in inverse video by specifically disabling character enhancements after printing A$.

Now look at how these functions might be used in a program. This small program segment defines two softkeys. One will cause program execution to continue, while one will terminate the program. The prompt requesting operator input will appear in the center of the display screen in inverse video.

```
1060 WIDTH 255
1070 PRINT FNHOME$;
1080 PRINT ESC$+"&jB"
1090 PRINT FNKEY$(1,"CONTINUE","PROCEED");
1100 PRINT FNKEY$(8,"EXIT TO  CP/M","EXIT");
1110 PRINT FNCURSOR$(10,20);
1120 PRINT FNIV$("CONTINUE?");
1130 PRINT FNCURSOR$(0,20);
1140 INPUT " ",A$
1150 IF A$="PROCEED" GOTO 2000
1160 IF A$="EXIT" GOTO 5000
2000 GOTO 1000
5000 STOP
5010 END
```

Remember, the semi-colon is used after each line to allow the programmer to position the cursor wherever necessary. BASIC will not perform any of the automatic 'carriage returns' as it would normally.

The sample functions provided above will not solve all of your terminal programming needs. However, it is hoped that by seeing some of the more useful escape sequences in use, you will be able to use the summary of sequences included in Appendix B. If more detail is necessary, refer to the "SERIES 100 System Reference Manual", which describes these sequences in much greater detail.

Index