Series 100/BASIC

HEWLETT PACKARD

Manual Part No.
45445-90007

Printed in U.S.A.
7/84

Table of Contents

Preface x
Manual Organization x
Notation Conventions xi

Chapter 1: Getting Started

The Series 100/BASIC User 1-1
Before You Begin 1-1
Starting BASIC 1-2
Modes of Operation 1-2
  Direct Mode 1-2
    Quick Computation 1-3
  Indirect Mode 1-3
    Line Format 1-4
Character Set 1-5
Entering a Program 1-8
Modifying a Program 1-9
  Edit Mode 1-9
    Edit Mode Subcommands 1-9
    Entering Edit Mode from a Syntax Error 1-14
  Modify Mode 1-14
    Using Modify Mode 1-15
    Start of Text Pointer 1-16
Error Messages 1-18

Chapter 1: Getting Started

Documenting Your Program 1-18
Printing Operations 1-19
  L Commands and Statements 1-19
Writing a Simple Program 1-20

Chapter 2: Data, Variables, and Operators

Introduction 2-1
Constants 2-2
  Single and Double Precision Form for Numeric Constants 2-3
Variables 2-3
  Variable Names and Declaration Characters 2-4
  Special Type Declaration Characters 2-4
    Reserved Words 2-4
    String Variables 2-5
    Numeric Variables 2-5
  Array Variables 2-6
Type Conversion 2-7
Expressions and Operators 2-10
  Arithmetic Operators 2-10
    Integer Division and Modulus Arithmetic 2-11
    Overflow and Division by Zero 2-12
  Relational Operators 2-12
  Logical Operators 2-13
  Functional Operators 2-16
  String Operations 2-17
    Concatenation 2-17
    Comparisons 2-17

Chapter 3: The BASIC Environment

Introduction 3-1
  BASIC 3-2

Chapter 4: File Operations

Disc Filenames 4-1
Disc Data Files—Sequential and Random Access 4-1
Sequential Files 4-1
Random Files 4-3
Creating a Random File 4-3
Accessing a Random File 4-4
Protected Files 4-8

Chapter 5: Programming Tasks

Introduction 5-1
System Commands 5-3
Using Commands as Program Statements 5-4
File Operations 5-5
Defining and Altering Data and Variables 5-6
Computer Control 5-7
Program Control, Branching, and Subroutines 5-8
Terminal Input and Output 5-10
Debugging Aids 5-11
BASIC Functions 5-12
General Purpose Functions 5-13
Input/Output Functions 5-13
Arithmetic Functions 5-14
Derived Functions 5-15
String Functions 5-16
Special Functions 5-17

Chapter 6: BASIC Statements, Commands, Functions, and Variables

Introduction 6-1
Chapter Format 6-2
ABS Function 6-3
ASC Function 6-3
ATN Function 6-3
AUTO Command 6-4
BLOAD Command/Statement 6-6
BSAVE Command/Statement 6-8
CALL Statement 6-9
CALLS Statement 6-10
CDBL Function 6-11
CHAIN Statement 6-12
CHR$ Function 6-17
CINT Function 6-17
CLEAR Statement 6-18
CLOSE Statement 6-20
COMMON Statement 6-21
CONT Command 6-23
COS Function 6-25
CSNG Function 6-25
CVI, CVS, CVD Functions 6-26
DATA Statement 6-27
DATE$ Function 6-28
DATE$ Statement 6-29
DEF FN Statement 6-30
DEF SEG Statement 6-32
DEF USR Statement 6-33
DEFINT/SNG/DBL/STR Statements 6-34
DELETE Command 6-36
DIM Statement 6-37
EDIT Command 6-38
END Statement 6-39

EOF Function 6-40
ERASE Statement 6-42
ERR and ERL Variables 6-43
ERROR Statement 6-45
EXP Function 6-47
FIELD Statement 6-48
FILES Command/Statement 6-50
FIX Function 6-51
FOR...NEXT Statement 6-52
FRE Function 6-55
GET Statement 6-56
GOSUB...RETURN Statement 6-57
GOTO Statement 6-59
HEX$ Function 6-60
IF Statement 6-61
INKEY$ Function 6-65
INP Function 6-65
INPUT Statement 6-66
INPUT# Statement 6-69
INPUT$ Function 6-71
INSTR Function 6-72
INT Function 6-73
KILL Command/Statement 6-74
LEFT$ Function 6-76
LEN Function 6-76
LET Statement 6-77
LINE INPUT Statement 6-78
LINE INPUT# Statement 6-79
LIST and LLIST Command 6-81
LOAD Command 6-83
LOC Function 6-84
LOF Function 6-84

LOG Function. 6-85
LPOS Function 6-85
LPRINT and LPRINT USING Statements. 6-86
LSET and RSET Statements 6-87
MERGE Command 6-88
MID$ Function. 6-90
MID$ Statement. 6-91
MKI$,MKS$,MKD$ Functions. 6-92
NAME Statement. 6-93
NEW Command. 6-94
NULL Statement 6-95
OCT$ Function. 6-96
ON ERROR GOTO Statement. 6-97
ON...GOSUB Statement 6-99
ON...GOTO Statement 6-100
OPEN Statement 6-101
OPTION BASE Statement 6-104
OUT Statement 6-105
PEEK Function. 6-106
POKE Statement 6-107
POS Function 6-108
PRINT Statement. 6-109
PRINT USING Statement. 6-112
PRINT# and PRINT# USING Statements. 6-117
PUT Statement. 6-120
RANDOMIZE Statement 6-121
READ Statement 6-123
REM Statement 6-125

Chapter 6: BASIC Statements, Commands, Functions, and Variables

RENUM Command . . . 6-127
RESET Command/Statement . . . 6-129
RESTORE Statement . . . 6-130
RESUME Statement . . . 6-131
RETURN Statement . . . 6-132
RIGHT$ Function . . . 6-133
RND Function . . . 6-133
RUN Command/Statement . . . 6-134
SAVE Command . . . 6-135
SGN Function . . . 6-136
SIN Function . . . 6-136
SPACE$ Function . . . 6-137
SPC Function . . . 6-137
SQR Function . . . 6-138
STOP Statement . . . 6-139
STR$ Function . . . 6-140
STRING$ Function . . . 6-140
SWAP Statement . . . 6-141
SYSTEM Command/Statement . . . 6-142
TAB Function . . . 6-143
TAN Function . . . 6-143
TIME$ Function . . . 6-144
TIME$ Statement . . . 6-145
TRON/TROFF Statements . . . 6-146
USR Function . . . 6-147
VAL Function . . . 6-148
VARPTR Function . . . 6-149
WAIT Statement . . . 6-151
WHILE...WEND Statement . . . 6-152
WIDTH Statement . . . 6-154
WRITE Statement . . . 6-155
WRITE# Statement . . . 6-156

Appendix A: Error Codes and Error Messages A-1

Appendix B: Using Terminal Features in BASIC

Introduction B-1
Sample Functions B-4

Appendix C: Reference Tables C-1

Appendix D: Assembly Language Subroutines

Introduction D-1
Memory Allocation D-2
CALL Statement D-3
USR Function D-8

Appendix E: Installing BASIC on the HP 110

Introduction E-1
Copying the Program Disc for Back-up E-2
Formatting the Back-up Disc E-3
Making the Back-up Copy E-4
Running Series 100/BASIC E-6
Running BASIC Using P.A.M. E-6
Running from an External Disc E-6
Running from the Electronic Disc E-7
Running BASIC Using MS-DOS E-8
Running from an External Disc E-8
Running from the Electronic Disc E-8

Appendix F: Installing BASIC on the HP 150

Introduction F-1
Making a Working Copy of BASIC F-1
  For Dual Disc Drive Users F-2
  For Hard Disc Drive Users F-4
Starting BASIC F-4

Index

Index I-1

Preface

This manual describes the version of Interpretive BASIC by Microsoft® that Hewlett-Packard supports. For a description of the compiled version of Microsoft® BASIC, you should consult the Microsoft® BASIC Compiler manual.

Manual Organization

Throughout this manual, the term "instruction" is a generic term that groups commands, statements, and functions under one name.

Chapter 1 introduces the Hewlett-Packard BASIC language and gives guidelines so you may start writing your own BASIC programs.

Chapter 2 describes general features about BASIC, such as data types and operations.

Chapter 3 gives specific information about the BASIC command line.

Chapter 4 describes files.

Chapter 5 groups the BASIC instructions together, according to the tasks that you may want to perform.

Chapter 6 is a comprehensive listing of all the BASIC commands, statements, functions, and variables. The listing is alphabetical.

The appendices provide further information on error codes and error messages, using your computer's terminal features, and assembly-language subroutines, as well as supplying necessary reference tables.

Notation Conventions

The notation conventions that we use in this manual adhere to the following rules:

CAPITAL LETTERS	You must enter those words that appear in capital letters exactly as they are shown. However, this only aids reading the syntax charts as BASIC automatically shifts variable names and key words to upper case letters.
lower case letters	Words shown in italicized, lower case letters are words that you must supply.
[square brackets]	Square brackets enclose items that are optional.
{braces}	Braces enclose multiple items when you must select between the available choices.
vertical bar |	A vertical bar divides the selection of items that are enclosed by braces.
ellipsis (...)	Items that are followed by an ellipsis may be repeated any number of times (up to the length of the input line).
punctuation	The punctuation symbols that serve special functions have been described above. You must include all other punctuation symbols (such as commas, semicolons, parentheses, quotation marks, etc.) exactly as they appear within the format charts.

Consider this example:

INPUT[;]["*prompt*" {; | ,}] *variable* [, *variable*]...

To be valid, an INPUT statement must contain the keyword INPUT and at least one variable. Since *variable* is italicized, you must replace this descriptive term with an appropriate name. Square brackets surround optional parameters. For example, the semicolon and prompt string are both optional. However, if you include a prompt, you must enclose the string in quotation marks and end the string with either a semicolon or a comma. You may list several variables, but you must separate them with commas.

Chapter 1

GETTING STARTED

The Series 100/BASIC User

To use Series 100/BASIC successfully, you only need to be familiar with general programming concepts and the BASIC language. If you are not familiar with BASIC, we recommend that you either read one of the introductory texts on programming in BASIC or take a beginning-level course on this language.

Before You Begin

If you have not yet installed the Series 100/BASIC software module into the software drawer in your computer, you need to do that now. Here's the procedure:

Step 1. If your computer's Edisc has any files that you don't want to lose, copy those files to an external disc. (To learn how to copy files from Edisc, refer to the section on "Copying Discs" in Chapter 5 of *Using the Portable PLUS*.)

Step 2. Install the BASIC software module in your HP 82982A Software Drawer. Just follow the *Software Module Installation Instructions* that came with your BASIC software package.

Step 3. Install the software drawer in a drawer receptacle on the bottom of your computer. Follow the *Drawer Installation Instructions* that came with your software drawer.

Starting BASIC

You start Series 100/BASIC just as you would any other application program on your Portable PLUS. First, go to the main P.A.M. screen on your computer. Move the display pointer over to the box labeled `BASIC` and then press `Start Applic` (f1).

Modes of Operation

Once you start BASIC, it shows you a symbol like this: `Ok`. This symbol, called a **prompt**, means that the BASIC interpreter is waiting for you to tell it what to do. This condition, where BASIC shows you a prompt and you respond, is called the **command level**. BASIC will remain at the command level until you enter a `RUN` command.

At the command level, you may converse with the BASIC interpreter in one of two modes: Direct Mode or Indirect Mode.

Direct Mode

In Direct Mode, you do not precede BASIC statements or functions with line numbers. Rather, you "talk" interactively with the BASIC interpreter, and it executes each instruction as you enter it.

For example,

```
Ok
PRINT "HELLO MOM" [Return]
HELLO MOM
Ok
```

Direct Mode is useful for debugging programs and for quick computations. You may use Direct Mode to display the results of mathematical and logical operations (using `PRINT` statements, as in the preceding example) or to store the results for later use (using the `LET` statement). However, the instructions that produce these results are lost after the interpreter executes the instruction.

Quick Computation

You may use BASIC as a calculator to perform quick calculations without writing a program. You can perform numeric operations in Direct Mode by entering a question mark (?) , then the expression. (BASIC interprets the question mark as an abbreviation for PRINT.) For example, to calculate two times the sum of four plus two where the sum is raised to the third power, type:

```
?2*(4+2)^3 [Return]
```

BASIC performs the calculation and prints the result:

```
432
```

When you assign values to variables with the LET statement, the values are not displayed. You can only view these values by printing them to the screen. Furthermore, the values that you assigned to variables are lost when you subsequently run a program or exit BASIC.

In the next example, the two LET statements set the value for X and Y. BASIC does not display these values. The last line is a PRINT statement that displays the answer for this simple problem.

```
LET X = 3 [Return]
LET Y = -8*X [Return]
PRINT ABS(X*Y) [Return]

72
```

Indirect Mode

You use Indirect Mode to enter programs. In this mode, you precede each line with a unique line number, and BASIC stores these lines in your computer's memory. You then execute the program by entering the RUN command.

For example,

```
Ok
10 PRINT "HELLO MOM" [Return]
RUN [Return]
HELLO MOM
Ok
```

Line Format

Program lines in a BASIC program have the following format:

nnnnn BASIC statement [**:***BASIC statement*]...

nnnnn represents a line number that may be from 1 to 5 digits in length. Permissible values range from 0 to 65529.

A program line always begins with a line number, may contain a maximum of 255 characters, and ends when you press the [Return] key. When a line contains more than 255 characters, BASIC truncates the excess characters.

Line numbers indicate the order in which BASIC stores the line in memory. They must be whole numbers. Numbers also serve as labels for branching and editing.

You may use a period with the `EDIT`, `LIST`, `AUTO`, and `DELETE` commands to refer to the current line. For example, `EDIT .` enables Edit mode on the last referenced or entered line.

A program line may contain a maximum of 255 characters. You may accomplish this in one of two ways. The simplier procedure is to type continuously, without pressing the [Return] key. In this case, if the line width remains at its default setting of 80 characters per line, a blank line appears in the program listing. It does not affect program execution. You can avoid printing the blank line by using the `WIDTH` statement to set the line width to 255.

However, if you want to "format" the line (for example, put the `THEN` and `ELSE` parts of an `IF` statement on separate lines), you may end a screen line by pressing [CTRL] [J]. This generates a line feed character which moves the cursor to the next screen line without terminating the logical line. A **logical line** is a string of text that BASIC treats as a unit. When you finish typing the logical line, pressing the [Return] key ends the line at that point.

NOTE

You must always end the last screen line of a logical ("program") line by pressing the [Return] key.

BASIC statement is any legal BASIC instruction.

A BASIC statement is either executable or non-executable. Executable statements instruct BASIC on what action it should undertake next. For example, `LET PI = 3.141593` is an executable statement. `DATA` and `REM` statements are non-executable statements. They result in no direct action by BASIC when BASIC encounters them.

You may enter multiple statements on one line, but you must separate each statement with a colon (`:`).

Character Set

The BASIC character set contains the alphabetic characters, numeric characters, and a selected set of special symbols.

Alphabetic characters are either upper-case or lower-case letters.

Numeric characters are the decimal digits 0 through 9.

Table 1-1 lists the special characters that BASIC supports.

Table 1-1. BASIC Special Characters

Character	Description
	Blank
=	Equal sign or assignment symbol
+	Plus sign or concatenation symbol
-	Minus sign
*	Multiplication sign or asterisk
/	Division sign or slash character
\	Integer division symbol or backslash
^	Exponentiation symbol or caret
%	Percent sign or integer type declaration character
!	Exclamation point or single-precision type declaration character
#	Number sign or double-precision type declaration character
$	Dollar sign or string type declaration character
(	Left parenthesis
)	Right parenthesis
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
;	Semicolon
:	Colon or program statement separator
&	Ampersand
?	Question mark
<	Lesser than symbol
>	Greater than symbol
@	At sign
_	Underscore
'	Apostrophe or remark delimiter
"	Quotation mark or string delimiter

BASIC also recognizes the following keyboard keys:

Key	**Function**
DEL	Deletes the last-typed character; also performs an automatic carriage return when all the characters on the line are deleted.
ESC	"Escapes" Edit mode subcommands.
Backspace	Backspaces over and deletes the last-typed character.
Tab	Moves the cursor to the next tab stop. (BASIC sets tab stops at every eighth column position beginning with the first column or at columns 1, 9, 17, and so on.)
Return	Serves several functions. These include terminating an input line and leaving Edit mode.

BASIC recognizes the following control characters:

CTRL A	Enters Edit mode on the line being typed.
CTRL C	Stops program execution and returns control to the BASIC command level.
CTRL G	Rings the computer's bell.
CTRL H	Backspaces over (and deletes) the last-typed character. (This duplicates the operation of the Backspace key.)
CTRL I	Moves the cursor to the next tab stop. (This duplicates the operation of the Tab key.)
CTRL J	Generates the line feed character.
CTRL O	Halts program output, but execution continues.
CTRL Q	Resumes program execution after it was suspending by a Control-S.
CTRL R	Prints the line that you are currently entering. (You might use this keystroke combination to "clean" a line of the highlighting characters produced by the DEL key.)
CTRL S	Suspends program execution.
CTRL U	Deletes the line that you are currently typing.

Entering A Program

You enter a program by simply typing the required text. As you type the characters over the keyboard, the editor interprets each keystroke. You may use this feature to reduce your typing. For example, the editor interprets a question mark (?) as the reserved word PRINT.

BASIC considers any line of text that begins with a number to be a BASIC statement. It then takes one of the following actions:

- adds a new line to the program if the line number doesn't currently exist
- replaces the line if the line number does exist
- deletes an existing line if requested to do so
- displays an error message if:
  - you attempt to delete a nonexistent line
  - program memory is exhausted

If BASIC prints a Direct Mode message on the screen, the editor automatically erases the message when you move the cursor to that line. This prevents the message from being entered as program text and producing syntax errors.

When you are using BASIC in Direct Mode, BASIC only recognizes those keys that were described previously. For example, you may delete a character on the line you are typing by pressing the [Backspace] key, the [DEL] key, or by simultaneously pressing the [CTRL] and [H] keys. If you attempt to "backspace" by using the cursor control keys, the characters are still transmitted to the BASIC interpreter. They are not deleted as you might expect.

When you delete characters by pressing the [DEL] key, BASIC surrounds the deleted text with backslashes (\). Pressing [CTRL] [H] has the same effect as pressing the [Backspace] key. After you delete any undesirable characters, you can continue typing the line from that point.

You may delete the line that you are currently typing by simultaneously pressing the [CTRL] and [U] keys. After it deletes the line, BASIC automatically performs a carriage return (moves the cursor to the beginning of that line.)

You may delete the program that is currently residing in computer memory by entering the NEW command. You normally use this command before you begin entering a new program.

Modifying A Program

The BASIC program editor is a "line" editor. That is, you can only modify one line at a time. You incorporate the changes into a line by pressing the [Return] key while the cursor is anywhere within that line.

NOTE

You need not move the cursor to the end of a logical line before you press the [Return] key. The editor "knows" where each line ends, and it processes the entire line, regardless of the cursor's position when you press the [Return] key.

You may choose between two methods to modify a line that currently resides in your computer's memory: retyping the line in its entirety, or entering Edit mode (by using the `EDIT` command). Additionally when running BASIC on the HP 150, you may use "Modify Mode" to edit text. (For details of this feature, see the discussion under Modify Mode.)

Edit Mode

Edit mode requires special one-character subcommands to edit a line. You enter Edit mode by typing the command `EDIT` and either a line number or a period (if you want to modify the last line). BASIC responds by displaying the line number of the specified line and a space character, then waits for you to enter a subcommand.

Edit Mode Subcommands

You may use Edit mode subcommands for either moving the cursor or performing edit operations. Edit operations include inserting or deleting text, replacing text, of searching for text within a line. The subcommands are not displayed. You may precede most of the Edit mode subcommands with an integer. This causes the command to be executed that number of times. When you omit the number, BASIC executes the subcommand once.

Edit mode subcommands may be categorized by the following functions:

- Moving the cursor
- Inserting text
- Deleting text
- Finding text
- Replacing text
- Ending and restarting Edit mode

Moving the Cursor

[Space bar]	Use the [Space bar] to move the cursor to the right. When you precede this action by a number, the cursor moves right that number of spaces.
[Backspace]	Use the [Backspace] key to move the cursor to the left. When you precede this action by a number, the cursor moves left that number of spaces.

Inserting Text

I

The I subcommand inserts text into a line. Any text you type after you enter Insert mode is inserted into the line.

You may end Insert mode by pressing the [ESC] key. Pressing the [Return] key moves the cursor to the beginning of the next line and ends both Insert mode and Edit mode.

While using the Insert (I) command, you may delete characters to the left of the cursor by pressing the [Backspace], [DEL], or [UNDERSCORE] key. Pressing the [Backspace] key repositions the cursor under the deleted character. Pressing either the [DEL] or [UNDERSCORE] key prints an underscore for each character you delete.

When you attempt to insert a character into a line and that character would make the line longer than 255 characters, BASIC rejects the character and rings the computer's bell.

X

The X subcommand extends a line. It moves the cursor to the end of the line, puts the keyboard into Insert mode, and then functions as if you had enter the Insert command (I). You may end this function by pressing either the [ESC] key or the [Return] key. (Pressing the [Return] key also terminates Edit mode.)

Deleting Text

`D` — The `D` subcommand deletes characters to the right of the cursor. To delete multiple characters, type the required number before you enter the `D` subcommand. BASIC echoes all deleted characters to the screen, with the deleted text surrounded by backslashes. BASIC positions the cursor to the right of the last character deleted. When there are fewer than the given number of characters to the right of the cursor, BASIC erases the remainder of the line.

`H` — The `H` subcommand deletes all characters to the right of the cursor and automatically enters Insert mode. You may find this subcommand useful for replacing text at the end of a line.

Finding Text

`S` — The `S` subcommand searches for a character. When you precede the subcommand with a number, BASIC searches for that occurrence of the character. For example, if you give the command:

```
5Sp
```

BASIC searches for the fifth occurrence of the letter "p". BASIC positions the cursor before the character when the search is successful. If the search fails, the cursor stops at the end of the line. BASIC displays all characters that it passes over while conducting the search.

`K` — The `K` subcommand resembles the `S` subcommand except that BASIC deletes all the characters it passes over while conducting the search. BASIC positions the cursor before the specified character, and it displays all deleted characters enclosed by backslashes.

Replacing Text

`C` — The `C` subcommand changes the next character in the line to the specified character. When you want to search for a specific occurrence of a character before changing it, precede the letter "`C`" with the appropriate number.

Ending and Restarting Edit Mode

[Return]	Pressing the [Return] key prints the remainder of the line, saves any changes you have made, and exits Edit mode.
`E`	The `E` subcommand saves any changes you have made and exits Edit mode.
`Q`	The `Q` subcommand exits Edit mode without saving any changes that you made to the line during Edit mode.
`L`	The `L` subcommand lists the remainder of the line, saves any changes that you made, and repositions the cursor at the beginning of the line. Edit mode remains active. (You usually would use this subcommand to list a line when you first enter Edit mode.)
`A`	The `A` subcommand restores the line to its original state (cancels any changes) and repositions the cursor at the beginning of the line so you can start again.
[Ctrl][A]	Simultaneously pressing the [Ctrl] and [A] keys takes you into Edit mode on the line that you are currently typing. BASIC executes a carriage return, prints an exclamation point (`!`) and a space, and positions the cursor at the first character in the line. You may now enter any Edit mode subcommand.

NOTE

If you have just entered a line and decide you want to edit it, just type `EDIT.`. BASIC takes you into Edit mode at that line. In this context, the period (`.`) is a special symbol that refers to the line you just entered.

When BASIC receives an unrecognizable command or illegal character while in Edit mode, it ignores the command and sends a Control-G ("Bell" character) to ring your computer's bell.

Entering Edit Mode From A Syntax Error

When BASIC encounters a syntax error while executing a program, it automatically takes you into Edit mode at the line that caused the error. For example:

```
10 K=2(4) [Return]
RUN [Return]
Syntax error in 10
Ok
10
```

When this happens, modify the line to correct the error and then either press [Return] or use the `E` subcommand to exit Edit mode. However, modifying the line this way destroys all variable values. If you want to preserve the variable values, first exit Edit mode with the `Q` subcommand. BASIC will go back to the command level where you can examine the variable values.

Modify Mode

You cannot use Modify mode with your Portable PLUS. This section applies only to the HP 150.

With the HP 150, you may use Modify mode to edit program lines with a minimum of typing:

- `LIST` the lines of the program you want to edit.
- Enter Modify mode (as described in the next subsection).
- Move the cursor to the first line you want to modify.
- Use the keyboard's character editing keys to modify the line.
- Press [Return] to store the edited line into memory.

NOTE

When a BASIC statement takes up more than one screen line (that is, you pressed [Ctrl][J] to insert a line feed character while entering the line), you cannot use Modify mode to edit that statement. You must use Edit mode instead.

Using Modify Mode

You access the Modify mode softkeys by pressing the [System] key.

The function key labels assume the following values:

device control	margins/ tabs/col	service keys	modes	enhance video	define fields	set time	config keys

Pressing function key [f4] or touching the modes softkey label assigns the following values to the function keys:

LINE MODIFY	MODIFY ALL	BLOCK MODE	REMOTE MODE	SMOOTH SCROLL	MEMORY LOCK	DISPLAY FUNCTNS	AUTO LF

After these softkey labels appear, you may select between one of two modify modes.

LINE MODIFY You select this mode by pressing function key [f1] or by touching the LINE MODIFY label. When this mode is active, an asterisk appears in the screen label. You may then use the keyboard edit keys to modify the line. Pressing the [Return] key enters the line into the program and simultaneously ends Line Modify mode.

MODIFY ALL You select this mode by pressing function key [f2] or by touching the MODIFY ALL label. When this mode is active, an asterisk appears in the screen label. The operation resembles Line Modify mode, except Modify All mode remains active until you explictly turn it off by again pressing function key [f2] or touching the MODIFY ALL label. (The asterisk disappears from the screen label.) Pressing the [Return] key does not end Modify All mode.

While in either Modify mode, you can use the cursor control keys to position the cursor. You can also use the editing keys [Insert char] and [Delete char] to modify existing program lines.

To delete a character, place the cursor under the character you wish to delete, then press [Delete char]. To delete multiple characters, you must press [Delete char] once for each character you wish to delete.

The [Insert char] key acts as a toggle switch. That is, alternate presses of this key turns Insert Character mode on then off. When Insert Character mode is active, the message `Ins Char` appears on the screen's Status Line (the bottom line of the display). While the keyboard is set for Insert Character mode, any character you type is inserted before the cursor's current position.

Control-C has no effect in Modify mode.

NOTE

You can only use the [Insert char] and [Delete char] keys while you are in Line Modify or Modify All mode. Pressing these keys at any other time produces unpredictable results.

CAUTION

NEVER use either of the Modify modes when the `AUTO` command is active. Furthermore, as the BASIC interpreter does not recognize the [Insert line] or [Delete line] keys, you must avoid using these keys while in Modify mode.

Start of Text Pointer

In Modify mode, pressing the [Return] key transmits all characters beyond the start-of-text pointer (or the start-column pointer if no start-of-text pointer exists) to the BASIC interpreter.

Initially, lines of text have no start-of-text pointer. A line of text acquires a start-of-text pointer under these conditions:

- the line that you are editing is at the bottom of the display (that is, it is the last line you entered).
- the line was entered from the keyboard and not transmitted from a host computer.
- the first character must be an alphanumeric character, the space character, a backspace, or a control character.

If all these conditions exist, the start-of-text pointer points to the first character in the line.

When no start-of-text pointer exists, transmission begins from the start-column pointer. You may assign a value to the start-column pointer in one of two ways:

(1) You may configure this value in the Terminal Configuration menu by following these steps:

Step 1. Press the [System] key. This displays the following function key labels:

device control	margins/ tabs/col	service keys	modes	enhance video	define fields	set time	config keys

Step 2. Press function key [f8] or touch the config keys screen label. This changes the softkey labels to the following values:

global config		port1 config	port2 config	terminal config	accesory config		

Step 3. Pressing function key [f5] or touching the terminal config softkey label displays the TERMINAL CONFIGURATION menu.

Step 4. This menu contains an entry for Start Column. It is normally set to "1". If you want to set another value, touch this field or press the [Tab] key until the cursor is positioned at this location. You may now type in the number you want for Start Column.

Step 5. Pressing function key [f1] or touching the SAVE CONFIG softkey label activates your selection. It also returns your screen to the state where you left it. If you decide to leave the menu in its current state, you can press function key [f8] or touch the config keys softkey label to remove the menu and return the screen to its last display.

(2) You may set the start-column pointer manually by following these steps:

Step 1. Press the [System] key. This displays the following function key labels:

device control	margins/ tabs/col	service keys	modes	enhance video	define fields	set time	config keys

Step 2. Press function key [f2] or touch the margins/tabs/col screen label. This changes the softkey labels to the following values:

START COLUMN	SET TAB	CLEAR TAB	CLR ALL TABS	LEFT MARGIN	RIGHT MARGIN	CLR ALL MARGINS	TAB = SPACES

Step 3. Pressing function key [f1] or touching the START COLUMN softkey label sets the value of the start-column pointer to the cursor's current position. (This requires your moving the cursor to the proper column before you set the value.)

Error Messages

When the BASIC interpreter detects a fatal error (that is, one that halts program execution), it prints an appropriate error message. Appendix A provides a complete list of error codes and their meanings.

Documenting Your Program

As a general rule for writing good programs in BASIC, we recommend that you include plenty of comment lines to document the program properly. See the REM statement for further information.

Printing Operations

You may choose between two methods for accessing a printer from BASIC. You may use the printer control softkeys or you may use the BASIC "L" commands and statements. Refer to your Owner's Manual for information on the printer control softkeys.

L Commands And Statements

The L commands and statements print to the MS-DOS general list device and are not affected by the printer control softkeys. The L commands are:

LLIST	Prints a program listing directly to the printer.
LPRINT	Prints information that is supplied by a program.
LPRINT USING	Formats information that is supplied by a program.

Writing A Simple Program

You need a working knowledge of several commands to start programming in BASIC. The following discussion treats these commands in their simplest form. They represent the rudimentary commands that you need to begin working with the BASIC interpreter.

AUTO	Generates line numbers automatically when you press the [Return] key. You may end this feature by simultaneously pressing the [CTRL] and [C] keys (Control-C).
LIST	Displays all or part of a program on the computer's screen.
DELETE	Removes a line or lines from a program.
RENUM	Resequences the lines in a program.
RUN	Executes a program.
SAVE	Stores 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	Clears the program that is currently stored in your computer's memory. This frees memory so you may use the area for other purposes, such as starting a new program.
SYSTEM	Leaves BASIC and returns system control to the operating system.
[CTRL] [C]	Stops execution and returns control to the BASIC command level.

The following steps lead you through a simple exercise where you use each of these commands.

Step 1. Go to the main P.A.M. screen on your computer.

Step 2. Move the display pointer to the box labeled: BASIC

Step 3. Press Start Applic (f1) and wait for the command level prompt: Ok.

Step 4. To start programming, type:

AUTO Return

This command tells BASIC to automatically prompt you with the next line number after you finish each line in your program. Notice how BASIC starts you with the first line number, 10.

Step 5. Now type the following short program:

```
10 FOR I = 1 TO 10 Return
20      PRINT I Return
30 NEXT I Return
40 PRINT "LOOP DONE, I ="; I Return
50 END Return
60
```

Step 6. Simultaneously press the [Ctrl] and [C] keys to stop the automatic line number prompt.

Step 7. Type:

RUN [Return]

The program prints the output from the program to your screen:

```
1
2
3
4
5
6
7
8
9
10
LOOP DONE, I=11
Ok
```

Step 8. To list your program on the screen, type:

LIST [Return]

BASIC shows you the listing:

```
10 FOR I=1 TO 10
20      PRINT I
30 NEXT I
40 PRINT "LOOP DONE, I="; I
50 END
Ok
```

Step 9. BASIC provides a variety of ways to modify an existing program. In this step, you will use the Edit mode subcommands to change the first line of the program so the loop counts back from 10 to 1.

- Type:

EDIT 10 [Return]

BASIC takes you into Edit mode on line 10.

- Move the cursor to the number 1 (after the equal sign) with the space command:

  8 [Space bar]

  (This assumes that you have used the same spacing as shown in the example.)

- Erase the remainder of the line and enter Insert mode by typing:

  H

- Complete the FOR statement by typing:

  10 TO 1 STEP -1 [Return]

Step 10. List the program by using the LIST command.

BASIC responds by printing:

```
10 FOR I = 10 TO 1 STEP -1
20     PRINT I
30 NEXT I
40 PRINT "LOOP DONE, I ="; I
50 END
Ok
```

Step 11. Use the RUN command to see how your changes have affected program execution.

The following display appears on your screen:

```
 10
 9
 8
 7
 6
 5
 4
 3
 2
 1
LOOP DONE, I = 0
Ok
```

Step 12. Delete line 40 by typing:

`DELETE 40` [Return]

`LIST` your program again and notice that BASIC has deleted line 40 from the program.

Step 13. If you wish to have the program lines in sequential order, renumber the lines by typing:

`RENUM` [Return]

Listing the program shows that the line numbers have been resequenced starting with 10 and incrementing by 10 at each step.

Step 14. You save your program by giving it a name so the system can retrieve it. For example, if you want to name the file `PROG1`, type:

`SAVE "PROG1"` [Return]

Since the name for the program is a character string, you must surround the name with quotation marks. Additionally, since you omitted any reference as to which drive should receive the file, BASIC stores the file on the currently active disc drive.
To save the program on a different disc, type:

`SAVE "n:PROG1"` [Return]

Here, `n:` names the disc drive that you selected. If you selected drive C, for example, the command appears as:

`SAVE "C:PROG1"`

BASIC supplies the MS-DOS file type `.BAS` for you. After it has successfully written your file to disc, BASIC responds with its `Ok` prompt.

Step 15. To see a listing of all the files on the default disc (including the one you just saved), type:

`FILES` [Return]

Step 16. If you want to delete your program file from the default disc, type:

`KILL "PROG1.BAS"` [Return]

NOTE

When using the `KILL` command, you must supply the file type `.BAS` as BASIC provides no default file extension for you.

Step 17. If you want to erase the program file from your computer's memory, type:

`NEW` [Return]

This clears the memory area for BASIC so you can enter a program or begin another application.

NOTE

Using the `NEW` command does not clear the file from your disc.

Step 18. When you are ready to leave BASIC and return control to the operating system, type:

`SYSTEM` [Return]

NOTE

Before exiting, be sure to `SAVE` your program if you wish to use it again.

Chapter 2

DATA, VARIABLES, AND OPERATORS

Introduction

This chapter discusses both data representation and also the mathematical and logical operators that BASIC provides.

Numeric values may be integers, single-precision numbers, or double-precision numbers. BASIC stores all numeric values in binary representation:

- Integers require two bytes of memory storage
- Single-precision numbers require four bytes of memory storage
- Double-precision numbers require eight bytes of memory storage

An integer value may be any whole number between -32768 and +32767.

BASIC stores single-precision numbers with 7 digits of precision (or 24 bits of precision), and prints up to seven digits, although only six digits may be accurate.

BASIC stores double-precision numbers with 17 digits of precision (or 56 bits of precision), and prints the number with up to 16 decimal digits.

Constants

The actual values that BASIC uses during program execution are called constants. Constants may be numeric values or string values.

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

```
"HELLO"
"Linda Kay"
"$75,000.00"
```

Numeric constants are positive or negative numbers. In BASIC, numeric constants never contain commas.

There are five types of numeric constants:

Integer constants Integer constants are whole numbers between -32768 and +32767. They never contain decimal points.

Fixed point constants Fixed point constants are positive or negative real numbers (that is, numbers that contain decimal points). For example, 1.0 is a fixed point constant; not an integer constant.

Floating point Floating point constants are positive or negative numbers that are given 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. For example,

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

(Double-precision floating point constants use the letter `D` instead of `E`, as in `235.988D7`.

Hex constants Hexadecimal numbers use a Base-16 numeric system. The letters `A` through `F` correspond to the numbers 10 through 15. You must prefix hexadecimal numbers with the symbols `&H`.
For example,

```
&HFF
&H32F
```

Octal constants

Octal numbers use a Base-8 numeric system. To signify an octal number, you must precede the number with an `&O` or `&`. For example,

```
&O347
&777
```

Single and Double Precision Form for Numeric Constants

A single-precision constant is any numeric constant that has:

- seven or fewer digits: `46.8`
- exponential form using `E`: `-1.09E-06`
- a trailing exclamation point (`!`): `3.141593!`

A double-precision constant is any numeric constant that has:

- eight or more digits: `345692811`
- exponential form using `D`: `-1.09432D-06`
- a trailing number sign (`#`): `3.141593#`

Variables

Variables are names that represent values within a BASIC program. You may explictly assign the value to a variable (for example, by using the `LET` statement). A variable may also obtain a value as the result of a computation (for example, `AREA = PI * RADIUS^2`). BASIC assumes all numeric variables have the value of zero and all string variables have the value of the null string until you actually assign them a value.

Variable Names and Declaration Characters

BASIC variable names may contain a maximum of 40 characters. Allowable characters are letters, the decimal digits, and a period. The first character must be a letter. The last character may be a type declaration character (either %, !, #, or $).

Examples of valid variable names are:

```
PAGELENGTH
SALES.1983
OUTER.LIMIT
```

BASIC would reject the following variable names:

A.HORRENDOUSLY.LONG.VARIABLE.NAME.FOR.THE.VALUE.OF.PAGELENGTH exceeds the limit of 40 characters.

1983SALES starts with a digit. The first character must be a letter.

OUTER LIMIT contains an embedded space.

Special Type Declaration Characters

BASIC recognizes several special type declaration characters and reserved words.

Reserved Words

Reserved words include all BASIC commands, statements, function names, and operator names. Appendix C provides a complete list of BASIC reserved words.

A variable name may not be a reserved word, but can contain embedded reserved words. For example, LOG and WIDTH are both BASIC reserved words, but LOG.WIDTH is a valid variable name.

BASIC assumes that a series of characters beginning with the letters FN is a call to a user-defined function. Therefore, you should never use these characters as the first two letters of a variable's name.

String Variables

You may designate string variable names with a dollar sign ($) as the last character, or you may declare them in a DEFSTR statement.

For example,

```
TITLE$
```

or

```
10 DEFSTR T
20 TITLE = "1983 Sales Report"
```

The dollar sign is a variable type declaration character. It "declares" that the variable represents a string. See Chapter 6 for a full discussion of the DEFSTR statement.

Numeric Variables

Numeric variable names may declare themselves to be integer, single-precision, or double-precision values. The type declaration characters for these variables names are:

%	Integer variable
!	Single-precision variable
#	Double-precision varialbe

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

Examples:

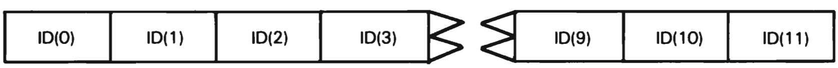
PI#	Declares PI to be a double-precision variable
MAX!	Declares MAX to be a single-precision variable
COUNT%	Declares COUNT to be an integer variable
LENGTH	Defaults to a single-precision variable

BASIC provides another method for declaring numeric variable types. This involves using the BASIC statements DEFINT to define integer variables, DEFSNG to define single-precision variables, and DEFDBL to define double-precision variables.

Array Variables

An array is a group of values (or a table) that you reference with a single variable name. The individual values in the array are called elements. You refer to each element by using the array's name and a subscript. The subscript may be an integer or an integer expression.

You declare an array by dimensioning it. You normally do this with the `DIM` statement. For example, `DIM ID$(11)` creates a one-dimensional, string array called `ID$`. Eleven is the index number for the "last" element of the array. When no `OPTION BASE` statement has executed, the "first" element of the array is `ID$(0)`. Therefore, this `DIM` statement creates an array of twelve elements. Each element is a variable-length string. An implicit act of declaring an array is assigning initial values for each array element. BASIC sets the elements of a string array equal to the null string (that is, the "empty" string or a string with zero length).



As another example, consider the statements:

```
OPTION BASE 1
DIM SALES(3,4)
```

These statements also create an array of twelve elements, but in this case the elements are grouped together in 3 rows of four columns each. (The columns could represent the four fiscal quarters of a year, and the rows could represent the years 1981 to 1983.) Since the array name has no type declaration character, BASIC sets the elements of the array to single-precision numbers and assigns the value of zero to each element.

SALES (1,1)	SALES (1,2)	SALES (1,3)	SALES (1,4)
SALES (2,1)	SALES (2,2)	SALES (2,3)	SALES (2,4)
SALES (3,1)	SALES (3,2)	SALES (3,3)	SALES (3,4)

An array variable name has as many subscripts as there are dimensions in the array. For example, when `OPTION BASE 1` is used, `VECTOR(10)` refers to the tenth value in a one-dimensional array, and `MATRIX(1,4)` refers to the fourth element in the first row of a two-dimensional array.

The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Type Conversion

When necessary, BASIC can convert a numeric constant from one type to another. The following examples illustrate the rules and operation of this automatic conversion.

1. When a numeric variable of one type is set equal to a numeric constant of a different type, BASIC stores the number as it was declared in the variable name. Trying to set a string variable equal to a numeric value, or vice versa, however, results in a `Type mismatch` error.

   Example:

   ```
   10 ROUND% = 23.42
   20 PRINT ROUND%
   30 ROUND% = 23.55
   40 PRINT ROUND%
   RUN
    23
    24
   ```

2. When evaluating an expression, BASIC converts all operands in an arithmetic or relational operation to the degree of precision of the most-precise operand. BASIC also calculates the result to this degree of precision.

   Consider these examples:

   a. BASIC performs the following calculation in double-precision arithmetic because the numerator is given as a double-precision number. BASIC also stores the result as a double-precision value.

```
10 TWO.THIRDS# = 2#/3
20 PRINT TWO.THIRDS#
RUN
 .6666666666666667
```

   b. BASIC performs the following calculation in double-precision arithmetic because the numerator is given as a double-precision number. Since the variable is a single-precision variable (by default), BASIC rounds the result and stores the value as a single-precision value.

```
10 TWO.THIRDS = 2#/3
20 PRINT TWO.THIRDS
RUN
 .6666667
```

   c. Logical operators convert their operands to integers and return an integer result. Operands must be in the range of -32768 to +32767, or an `Overflow` error occurs.

```
10 FALSE = 0
20 PRINT FALSE
30 PRINT NOT FALSE
40 TRUE = 99.44
50 PRINT NOT TRUE
60 PRINT TRUE AND FALSE
RUN
 0
-1
-100
 0
 Ok
```

d. When a floating point value is converted to an integer, BASIC rounds the fractional portion.

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

```
10 COMPROMISE% = 55.44
20 PRINT COMPROMISE%
RUN
 55
```

e. When you assign a single-precision value to a double-precision variable, only the first seven digits, rounded, of the converted number are valid. This happens 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 is less than 6.3E-8 times the original single-precision value. For example,

```
10 PI = 3.141593
20 BADPI# = PI
30 PRINT PI, BADPI#
RUN
  3.141593   3.141592979431152
```

Expressions and Operators

An expression may be a string or numeric constant, or a variable; or it may be a combination of constants and variables with suitable operators to produce a single value.

Operators perform mathematical or logical operations on values. BASIC provides the following four categories of operators:

- Arithmetic
- Relational
- Logical
- Functional

Arithmetic Operators

Table 2-1 lists the arithmetic operators.

Table 2-1. BASIC Arithmetic Operators

Operator	Operation	Sample Expression
^	Exponentiation	RADIUS^2
-	Negation	-DEBITS
*	Multiplication	BASE * HEIGHT
/	Point Division	AREA / PI
+	Addition	WAGES + DIVIDENDS
-	Subtraction	INCOME - TAXES

BASIC evaluates an expression based upon the order of precedence of the included operators. Exponentiation is evaluated first, followed by negation. Next, any multiplication or division is performed, and finally, all addition or subtraction operations are performed. In the case of multiple operators with equal precedence, BASIC evaluates the expression from left to right.

You may change the order of evaluation by using parentheses. BASIC first evaluates all operations within parentheses. (Within a parentheses grouping, the order precedence shown above is maintained.) Consider these examples:

Without parentheses: `4^3^2 = 4096`

With parentheses: `4^(3^2) = 262144`

The following expanded version of the first example uses parentheses to show the implicit grouping of operations by supplying all parentheses.

```
((4^3)^2) = (64)^2 = 4096
```

The following list shows how you would write algebraic expressions in BASIC.

Algebraic Expression	BASIC Expression
$X + 2Y$	`X + 2 * Y`
$X - \frac{Y}{Z}$	`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)`

NOTE

You must always separate two consecutive operators by parentheses.

Integer Division and Modulus Arithmetic

You specify the integer division operation with a backslash (\). With integer division, BASIC rounds the operands to integers before it performs the division. It then truncates the quotient to an integer value. (The operands must be within the range -32768 to +32767.) For example,

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

In the order of precedence, integer division follows multiplication and floating point division.

You specify modulus arithmetic with the `MOD` operator. The `MOD` operator returns the remainder from an integer division operation. For example,

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

The precedence of modulus arithmetic is just after integer division.

Overflow and Division by Zero

When BASIC is evaluating an expression, if it encounters an zero divisor, it displays a `Division by zero` error message, sets the result to machine infinity with the sign of the numerator, and continues program execution. If the evaluation of an exponentiation results in zero being raised to a negative power, BASIC again displays the `Division by zero` error message, sets the result to positive machine infinity, and continues program execution.

When BASIC encounters a number whose absolute value is too large for it to store, it displays the `Overflow` error message, sets the result to machine infinity with the appropriate sign, and continues program execution.

Machine infinity is approximately equal to 1.7 * 10^38.

Relational Operators

Relational operators compare values or variables. The result of the comparison is either "true" (-1) or "false" (0). You may use this result to control the flow of a program. (See the description of the `IF` statement.)

Table 2-2 summarizes the relational operators.

Table 2-2. BASIC Relational Operators

Operator	Relation	Sample Expression
=	Equality	`COUNTER = LIMIT`
<>	Inequality	`LENGTH <> HEIGHT`
<	Less than	`COLUMN < 80`
>	Greater than	`ROW > 24`
<=	Less than or equal to	`YEAR <= 1984`
>=	Greater than or equal to	`LINECOUNT >= PAGESIZE`

You may also use the equal sign to assign a value to a variable. (See the description of the `LET` statement.)

When arithmetic and relational operators are combined in one expression, BASIC performs all arithmetic operations first. For example, the expression:

```
TMARGIN + BMARGIN + LINECOUNT <= PAGESIZE/2
```

is true when the sum of `TMARGIN`, `BMARGIN`, and `LINECOUNT` is less than or equal to half the `PAGESIZE`.

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result that is either **true** (not zero) or **false** (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of the logical operators are summarized in the following **truth tables**. The operators are listed in their order of precedence.

NOT

Purpose: `NOT` inverts its operand. That is, a true bit is set to false and a false bit is set to true.

Truth Table:

X	NOT X
1	0
0	1

AND

Purpose: `AND` requires both operands to be true if the result is to be true.

Truth Table:

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

OR — Inclusive OR

Purpose: `OR` returns true when either operand or both operands are true.

Truth Table:

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

XOR — Exclusive OR

Purpose: `XOR` returns true when either operand is true.

Truth Table:

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

IMP — Implied

Purpose: `IMP` returns true when both operands are the same. If they differ, the result is the same as the second operand.

Truth Table:

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

EQV — Equivalent

Purpose: EQV returns true when both operands have the same value.

Truth Table:

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 value that determines program flow. For example,

```
IF VALUE < 0 OR VALUE > 100 THEN 480
IF QUARTER < 4 AND YEAR = 1983 GOTO 1000
IF NOT LIMIT THEN 100
```

Logical operators convert their operands to sixteen bit, signed, two's-complement integers in the range -32768 to +32767. (If either operand is outside this range, an error occurs.) When both operands are given as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, that is, each bit of the result is determined by the corresponding bits in the two operands.

You may use logical operators to test bytes for a particular bit pattern. For instance, you may use the AND operator to **mask** all but one of the bits of a status byte. Similarly, you may use the OR operator to **merge** two bytes to create a particular binary value.

The following examples demonstrate how you may use the logical operators in this fashion. (Each number is represented in two bytes, or 16 bits; however, the examples ignore all leading zeros.)

Operation	Calculation
`63 AND 16 = 16`	63 is binary `111111` and 16 is binary `10000` so `111111 AND 10000` is `10000` (or 16).
`15 AND 14 = 14`	15 is binary `1111` and 14 is binary `1110` so `1111 AND 1110` is `1110` (or 14).
`-1 AND 8 = 8`	-1 is binary `1111111111111111` and 8 is binary `1000` so `1111111111111111 AND 1000` is `1000` (or 8).
`4 OR 2 = 6`	4 is binary `100` and 2 is binary `10` so `100 OR 10` is `110` (or 6).
`10 OR 10 = 10`	10 is binary `1010`, so `1010 OR 1010` is `1010` (or 10).
`-1 OR -2 = -1`	-1 is binary `1111111111111111` and -2 is binary `1111111111111110` so `1111111111111111 OR 1111111111111110` is `1111111111111111` (or -1).
`TWOCOMP = (NOT X)+1`	The two's-complement of any integer is the bit complement plus one. For example, if `X` is equal to 2, `NOT X` would be binary `1111111111111101`. This is decimal -3, and -3 plus 1 is -2, or the complement of 2.

Functional Operators

A function is a predetermined operation that performs the specified task on its operand. BASIC has "intrinsic" functions that reside in the system, such as `SQR` (square root) or `SIN` (sine).

BASIC also allows "user-defined" functions that you write. See the `DEF FN` statement for further details.

String Operations

BASIC provides two string operations. These operations are string concatenation and string comparisons. (See the section on "String Functions" in Chapter 5 for a listing of the built-in functions that manipulate strings.)

Concatenation

You can join strings together (concatenate them) by using the plus sign (+). For example,

```
10 A$ = "File" : B$ = "Name"
20 PRINT A$ + B$
30 PRINT "Another " + A$ + B$
RUN
FileName
Another FileName
```

Comparisons

You can compare strings by using the same relational operators that you use for numeric comparisons:

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

BASIC compares strings by taking one character at a time from each string and comparing their ASCII codes. When all the ASCII codes are the same, the strings are equal. When the ASCII codes differ, the lower code number precedes the higher number. If, during a string comparison, BASIC reaches the end of one string while characters still remain in the other, the shorter string is said to be smaller. Leading and trailing blanks are significant. For example,

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"FILENAME" <> "filename"
"kg" > "KG"
"123 " > "123"
"SMYTH" < "SMYTHE"
B$ < "52/4/24" (where B$ = "47/5/10")
```

You may use string comparisons to test string values or to alphabetize strings. When using string constants in comparison expressions, you must enclose the constant in quotation marks.

Chapter 3

THE BASIC ENVIRONMENT

Introduction

Chapter 1 describes the easiest procedure for running BASIC on your computer. However, entering BASIC through an MS-DOS system command gives you added flexibility in establishing the BASIC environment.

This chapter describes `BASIC`, the MS-DOS system command that you must you use to enter BASIC.

BASIC

Format:

`BASIC`

`BASIC` [*filename*]
[`/F:`*numfiles*] [`/S:`*recl*]
[`/M:`*highest.mem.loc*]

Purpose: Loads the BASIC interpreter program into your computer's memory.

Remarks: *filename* directs BASIC to run the specified BASIC program immediately. You may use this parameter to run programs in batch mode by including the filename in the command line of a `.BAT` file (such as `AUTOEXEC.BAT`). You must end each program with a `SYSTEM` statement. This allows the next command from the `.BAT` file to execute.

`/F:` sets the number of files that you can open simultaneously. Each file requires 62 bytes for the File Control Block (FCB) and 128 bytes for the data buffer. You may alter the size of the data buffer with the `/S` option switch. When you omit the `/F` parameter, BASIC sets the value to 3.

The number of open files that MS-DOS supports depends upon the value of the `FILES=` parameter in the `CONFIG.SYS` file. When you are using BASIC, we recommend that you set the `FILES` parameter to `10`. BASIC allocates the first three files to Stdin, Stdout, Stderr, Stdaux, and Stdprn, then it sets aside an additional file for `LOAD`, `SAVE`, `CHAIN`, `NAME`, and `MERGE` commands. When you set `FILES=10`, six files remain for BASIC input/output files. Thus, `/F:6` is the maximum number of files that you may request when `FILES=10` appears in the `CONFIG.SYS` file. Attempting to open a file after all the file handles have been taken results in a `Too many files` error message.

/S: sets the maximum record size for random-access files to *recl*. When you omit this parameter, BASIC sets the value to 128 bytes.

NOTE

The record size option for the OPEN statement cannot exceed this value.

/M: sets the highest memory location that BASIC uses. Normally, BASIC allocates 64K bytes of memory for the Data and Stack segment. When you omit this parameter or set it to zero, BASIC attempts to allocate as much memory as it can, up to a maximum of 65536 bytes.

NOTE

You may use decimal, octal, or hexadecimal numbers for *numfiles*, *highest.mem.loc*, and *recl*. You must precede octal numbers with &O, and hexadecimal numbers with &H.

Examples:

The first example uses the default settings. Thus, it uses 64K of memory, permits 3 opened files, then loads and executes PAYROLL.BAS:

```
A> BASIC PAYROLL
```

The second example also uses 64K of memory but permits 6 opened files. It loads and executes INVENT.BAS:

```
A> BASIC INVENT/F:6
```

The next example uses the first 32K bytes of memory. The memory above 32K is free for the user:

```
A> BASIC /M:32767
```

The last example uses 4 files and sets a maximum record length of 512 bytes:

```
A> BASIC /F:4/S:512
```

Chapter 4

FILE OPERATIONS

Disc Filenames

Disc filenames obey the standard MS-DOS naming conventions. (Refer to your Owner's Guide.) All filenames may include a letter and a colon as the first two characters to specify a disc drive. For example, `A:` refers to drive A. If you omit this special symbol combination, BASIC assumes that all files refer to the currently selected disc drive. When you use either the `LOAD`, `SAVE`, `MERGE`, or `RUN` statements, BASIC attaches the file type extension `.BAS` to the filename if the filename is less than 9 characters long and you omitted a file extension. (No "`.`" appears in the filename.)

Disc Data Files—Sequential and Random Access

You may create two different types of disc data files for a BASIC program to access. They are sequential files and random access files.

Sequential Files

Sequential files have a simpler structure than random-access files, but they are limited by their flexibility and their speed of accessing data. When you write data to a sequential file, BASIC writes the information to the file in sequential order, one item after the other, in the order that it is sent. BASIC reads back the information in the same way.

You may use the following statements and functions with sequential files:

```
CLOSE
EOF
INPUT#
LINE INPUT#
LOC
OPEN
PRINT#
PRINT# USING
WRITE#
```

You must follow these steps to create a sequential file, then access its data:

Step 1. Open the file in `O` mode. For example,

```
OPEN "O",#1,"DATA"
```

Step 2. Write data to the file using the `PRINT#` or `WRITE#` statement. For example,

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

Step 3. To access the data in the file, you must close the file then reopen it in `I` mode. For example,

```
CLOSE #1
OPEN "I", #1, "DATA"
```

Step 4. Use the `INPUT#` statement to read data from the sequential file into the program. For example,

```
INPUT #1, X$,Y$,Z$
```

A program that creates a sequential file can also write formatted data to the disc with the `PRINT# USING` statement. For example, you could use the following statement to write numeric data to disc without using explicit delimiters:

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

In this example, the comma at the end of the format string (before the closing quotation mark) separates the items in the disc file.

Random Files

It takes more programming steps to create and access random files than sequential files. However, you may find the advantages of random-access files outweigh the time required to enter the extra steps.

With random files, BASIC stores and accesses information in distinct units called **records**. Since each record is numbered, you may access data anywhere in the file without reading through the file sequentially.

You may use the following statements with random-access files:

```
CLOSE
CVI  CVS  CVD
FIELD
GET
LOC
LSET/RSET
MKI$  MKS$  MKD$
OPEN
PUT
```

Creating a Random File

You must follow these steps to create a random file:

Step 1. Open the file for random access (R mode). The following example sets a record length of 32 bytes. When you omit the record length parameter, BASIC uses 128 bytes as the default record size.

```
OPEN "R", #1,"FILE",32
```

NOTE

The maximum logical record number is 32767. Theoretically, if you set the record size to 256 bytes, you may access files up to 8 megabytes in size.

Step 2. Use the FIELD statement to allocate space in the random file buffer for the variables that you plan to write to the random file. For example,

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```

Step 3. Use `LSET` to move the data into the random file buffer. Before you can place numeric values into this buffer, you must convert these values to strings by using one of the following functions:

`MKI$`	Converts an integer value to a string
`MKS$`	Converts a single-precision value to a string
`MKD$`	Converts a double-precision value to a string

Examples of the `LSET` statement are:

```
LSET N$ = X$
LSET A$ = MKS$(AMT)
LSET P$ = TEL$
```

Step 4. Write the data from the buffer to the disc using the `PUT` statement:

```
PUT #1, CODE%
```

Accessing a Random File

You must follow these steps to access the data in a random-access file:

Step 1. Open the file for random access (`R` mode). For example,

```
OPEN "R", #1, FILE#, 32
```

Step 2. Use the `FIELD` statement to allocate space in the random file buffer for the variables that you plan to read from the file. For example,

```
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 usually use one `OPEN` statement and one `FIELD` statement.

Step 3. Use the `GET` statement to move the desired record into the random file buffer. In the following example, `CODE%` contains the record number.

```
GET #1, CODE%
```

Step 4. Your program may now access the data in the buffer. However, numeric values must be converted from strings back to numbers. You do this with the convert functions:

CVI	Converts the data item to an integer
CVS	Converts the data item to a single-precision value
CVD	Converts the data item to a double-precision value

For example:

```
PRINT CVS(A$)
```

In the following example, the user accesses the random file called FILE by entering a 2-digit code at the keyboard. The program then reads the information that is associated with the code and displays it on the computer screen.

```
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 IF CODE% = 0 THEN CLOSE 1 : END
50 GET #1, CODE%
60 PRINT N$
70 PRINT USING "$$###.##"; CVS(A$)
80 PRINT P$ : PRINT
90 GOTO 30
```

The following program illustrates random file access. In this program, the record number serves as the part number. (It is assumed that the inventory never contains more than 100 different part numbers.) Lines 900 through 960 initialize the data file by writing `CHR$(255)` as the first character of each record. Later, lines 270 and 500 use this character to determine whether an entry already exists for that part number.

```
110 OPEN "R",#1,"INVEN.DAT",39
120 FIELD #1, 1 AS F$, 30 AS D$,
    2 AS Q$, 2 AS R$, 4 AS P$
130 PRINT : PRINT "FUNCTIONS:" : PRINT
140 PRINT 1, "INITIALIZE FILE"
150 PRINT 2, "CREATE NEW ENTRY"
160 PRINT 3, "DISPLAY INVENTORY FOR ONE PART"
170 PRINT 4, "ADD TO STOCK"
180 PRINT 5, "SUBTRACT FROM STOCK"
190 PRINT 6, "DISPLAY ALL ITEMS
    BELOW REORDER LEVEL"
200 PRINT 7, "END PROGRAM"
210 PRINT : PRINT : INPUT "FUNCTION"; FUNCTION
220 IF (FUNCTION <1) OR (FUNCTION > 7)
    THEN PRINT "BAD FUNCTION NUMBER" : GOTO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680,860
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$(Q%)
330 INPUT "REORDER LEVEL", R%
340 LSET R$ = MKS$(R%)
350 INPUT "UNIT PRICE"; P
360 LSET P$ = 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$(Q%)
660 PUT #1, PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
700     GET #1, I
710     IF ASC(F$) = 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
860 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
```

Protected Files

If you wish to save a program in a special binary format, you must use the "Protect" (P) option with the SAVE command. For example, the following statement saves the program named ETERNAL so it cannot be listed or edited:

```
SAVE "ETERNAL", P
```

As no command exists to "unprotect" the file, you may also want to save an unprotected copy of the program that you can list and change.

Chapter 5

PROGRAMMING TASKS

Introduction

When programming, you normally have a specific task that you wish to perform. The experienced programmer has no difficulty determining which BASIC instruction is appropriate for the task at hand. However, if some features of the language are new to you, you may have trouble isolating the best instruction. This chapter groups the various BASIC commands, statements, and functions into task-oriented areas. For example, if you want to review a document, you may know that you need an "output" statement, but you may not know which one. By looking under the terminal input and output section in this chapter, you would discover that BASIC provides five "printing" statements: `PRINT`, `LPRINT`, `PRINT USING`, `LPRINT USING`, and `WRITE`. You can get an indictation of each statement's use by reading its general description. Then you should consult Chapter 6 for full details on using the statement that you selected.

This chapter contains the following sections:

- System commands
- Using system commands as program statements
- File operations
- Defining and altering data and variables
- Computer control
- Program control, branching, and subroutines
- Terminal input and output
- Debugging aids
- General purpose functions
- Input/Output functions
- Arithmetic functions
- Derived arithmetic functions
- String functions
- Special functions

System Commands

System Commands are those commands that you enter on the BASIC command line and/or those that return control to the command line. The following list summarizes the system commands that BASIC provides.

`AUTO`	Automatically generates line numbers for program entry.
`BLOAD`	Loads the specified memory image file into your computer's memory.
`BSAVE`	Saves the contents of the specified area of memory to a disc file.
`CONT`	Continues program execution after you type a Control-C, or your program executes a `STOP` or `END` statement.
`DELETE`	Removes the specified lines from a BASIC program.
`EDIT`	Enables Edit mode on the specified line.
`FILES`	Lists the names of the files residing on a specified disc.
`KILL`	Deletes one or more files from a specified disc.
`LIST` and `LLIST`	Lists all or part of the program that is currently stored in memory to either the computer screen or a printer.
`LOAD`	Loads a BASIC program file from disc into memory.
`MERGE`	Incorporates statements contained in the specified disc file into the program that is currently stored in computer memory.
`NEW`	Deletes the program that is currently stored in computer memory and clears all variables.
`RENUM`	Renumbers the lines of a program so they occur in a specified sequence.
`RESET`	Closes all disc files and prints the directory information to every disc with open files.
`RUN`	Executes the program that is currently stored in your computer's memory.
`SAVE`	Saves the program currently stored in computer memory to a specified disc file.
`SYSTEM`	Exits BASIC and returns control to the operating system.

Using Commands as Program Statements

You may use several of the BASIC commands as program statements. Refer to the preceding discussion for each of the commands, then consult this section for its use within a program.

BLOAD	Programmatically loads code or data into a given area of memory.
BSAVE	Programmatically copies code or data from memory to a specified disc file.
FILES	Programmatically lists directory information.
KILL	Programmatically deletes the specified disc files.
RESET	Programmatically closes all disc files and prints the directory information to every disc with open files. (You should use this statement in any program that performs disc access.)
RUN	Programmatically re-executes a program from a specified line.
SYSTEM	Programmatically exits BASIC.

File Operations

BASIC provides the following instructions or handling files and their contents.

CLOSE	Concludes all input/output to a disc file.
EOF	Returns end-of-file for sequential and random-access files.
FIELD	Allocates space for variables in a random file buffer.
GET	Reads a record from a random disc file into a random file buffer.
INPUT#	Reads values from a sequential disc file and assigns them to program variables.
LINE INPUT#	Reads an entire line (up to 254 characters) from a sequential disc file and assigns the line to a string variable.
LOC	Returns the last record number in a GET or PUT statement.
LOF	Returns the length of the file in bytes.
LSET and RSET	Moves data from memory into a random file buffer in preparation for a PUT statement.
NAME	Changes the name of a disc file.
OPEN	Allows access to a file for either reading and/or writing.
PRINT# and PRINT# USING	Writes data to a sequential disc file.
PUT	Writes a record from a random file buffer to a random disc file.
WIDTH	Sets the printer line width by specifying the number of characters per line.
WRITE#	Writes data to a sequential file.

Defining and Altering Data and Variables

BASIC provides several statements that you may use within a program to define and manipulate data, variables, expressions, and arrays. The following list summarizes these statements.

CLEAR	Sets numeric and string variables to zero or null, closes all files, and optionally sets the end of memory and the amount of stack space.
COMMON	Passes variable values to a chained program.
DATA	Stores data for later access by a program's READ statements.
DEFINT/DEFSNG DEFDBL/DEFSTR	Declares that BASIC should automatically treat certain variable names as integer, single-precision, double-precision, or string variable types.
DIM	Sets the maximum values for an array's subscripts, allocates storage, and assigns an initial value to array elements.
ERASE	Removes an array from a program.
LET	Assigns the value of an expression to a variable.
MID$	Replaces a portion of one string with another string.
OPTION BASE	Determines if the minimum value for an array subscript should be zero or one.
READ	Reads values from a DATA statement and assigns them to variables.
RESTORE	Permits a program to reread DATA statements from a specified line.
SWAP	Exchanges the values of two variables.

Computer Control

Several BASIC statements let you control your computer from the program. These statements are:

DATE$	Sets the current date.
INP	Returns a byte, which is read from a microprocessor port.
OUT	Sends a byte to the microprocessor port.
POKE	Writes a byte into a memory location.
TIME$	Sets the current time.
WAIT	Suspends program execution while monitoring the status of a microprocessor input port.

To control your computer from the program, you can also use escape sequences. For example, the sequence ESC H "homes" the cursor, and the sequence ESC J clears the screen from the cursor to the end. Therefore, you could clear the entire display by executing this statement:

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

(CHR$(27) is the ASCII code for the escape character.)

For details on using escape sequences, refer to Appendix B. For a complete list of escape sequences that you can use with your Portable PLUS, refer to the *Portable PLUS Technical Reference Manual* (HP 45559K), which is available from your HP sales representative.

Program Control, Branching, and Subroutines

Several BASIC statements let you control the flow of your program through branching to other lines, subroutines, and programs. These statements are:

CALL	Calls an assembly-language subroutine.
CALLS	Calls a subroutine with segmented addresses.
CHAIN	Calls a program and passes variable values to it from the current program.
DEF FN	Names and defines a user-written function.
DEF SEG	Assigns the current segment address. Subsequent CALL, CALLS, POKE, PEEK, or USR instructions refer to this address.
DEF USR	Assigns the starting address of an assembly-language subroutine.
END	Ends program execution, closes all files, and returns control to the command level.
FOR...NEXT	Loops through a series of instructions a given number of times.
GOSUB...RETURN	Branches to and returns from a subroutine.
GOTO	Branches unconditionally to the specified line number.
IF	Determines program flow based on the result returned by a logical expression.
ON ERROR GOTO	Enables error trapping and specifies the first line number of the error-handling subroutine.
ON...GOSUB	Branches to a subroutine, or subroutines, depending on the value returned by the governing expression.
ON...GOTO	Branches to one of several specified line numbers, depending on the value returned by the governing expression.
RESUME	Continues the program after an error recovery procedure.

RETURN	Returns control to the next statement in a program after a GOSUB or an ON GOSUB statement.
STOP	Suspends program execution and returns control to the BASIC command level.
WHILE...WEND	Loops through a series of statements as long as a given condition is true.

You may divide the branching and subroutine statements into the following categories:

Unconditional branching:

- GOTO
- ON ... GOTO

Conditional branching:

- IF ... THEN [... ELSE]
- IF ... GOTO
- ON ERROR GOTO
- WHILE ... WEND

Branching to another program:

- CHAIN

Looping:

- FOR ... NEXT
- WHILE...WEND

Subroutines:

- CALL
- CALLS
- DEF FN
- DEF SEG
- DEF USR
- GOSUB ... RETURN
- ON ... GOSUB
- RETURN

Terminal Input and Output

You may use BASIC Input statements for entering information into programs from either the keyboard, disc files, or the `DATA` statement. You may use BASIC Output statements to copy information to the computer screen, a printer, a file, and/or a memory location. The following list summarizes these statements.

`INPUT`	Takes input from the keyboard.
`LINE INPUT`	Enters an entire line (up to 254 characters) to a string variable, without the use of delimiters.
`LPRINT` and `LPRINT USING`	Prints data to a line printer.
`NULL`	Sets the number of nulls to be printed at the end of each line. This applies to both the display and the printer.
`PRINT`	Prints data to the computer screen.
`PRINT USING`	Uses a specified format to print strings or numbers.
`WIDTH`	Sets the printer line width by specifying the number of characters per line.
`WRITE`	Writes data to the computer screen.

Debugging Aids

You use debugging statements to trace program execution, to define error codes, or to simulate error conditions. Since well-documented programs help prevent errors, we treat the `REM` statement as a debugging aid.

The following list summarizes the debugging statements that BASIC provides.

`ERROR`	Simulates the occurrence of a BASIC error; or allows you to define error codes.
`REM`	Inserts explanatory remarks into a program.
`TRON/TROFF`	Traces the execution of program statements.

BASIC Functions

BASIC provides several intrinsic functions. You may call these functions, without further definition, from any point in a program.

You must enclose a function's argument(s) in parentheses. Most function formats abbreviate the arguments as follows:

x and *y*	Represent numeric expressions
i and *j*	Represent integer expressions
x$* and *y$	Represent string expressions

If you give a function a floating point value when the function takes an integral argument, BASIC rounds the fractional portion and uses the integer result.

NOTE

The results that the BASIC interpreter returns to function calls are either integer, single-precision, or string values. Only the BASIC compiler returns double-precision values.

You may divide the functions into five general categories. These categories are:

- General Purpose Functions
- Input/Output Functions
- Arithmetic Functions
- String Functions
- Special Functions

General Purpose Functions

BASIC provides the following general-purpose functions:

DATE$	Returns the current date.
TIME$	Returns the current time.

Input/Output Functions

The Input/Output functions send or return information to the computer or a printer.

CVI,CVS,CVD	Convert string values to numeric values.
EOF	Returns end-of-file for sequential and random-access files.
INKEY$	Returns a one-character or null string from the computer's keyboard.
INPUT$	Returns a string from either the keyboard or a disc data file.
LOC	Returns the last record number in a GET or PUT statement.
LOF	Returns the length of the file in bytes.
LPOS	Returns the current position of the printer print head within the printer buffer.
MKI$,MKS$,MKD$	Convert numeric values to string values.
POS	Returns the print head's column position.
SPC	Prints spaces (blank characters) on the display.
TAB	Moves to a specified position on a line.

Arithmetic Functions

The RANDOMIZE statement and the arithmetic functions manipulate numeric expressions.

ABS	Returns the absolute value of the numeric expression.
ATN	Returns the arctangent of a numeric expression which you must give in radians.
CDBL	Converts a numeric expression to a double-precision number.
CINT	Converts a numeric expression to an integer by rounding off the fractional part.
COS	Returns the cosine of a numeric expression which you must give in radians.
CSNG	Converts a numeric expression to a single-precision number.
EXP	Returns *e* (where *e* = 2.71828...) to the power of *X*. *X* must be less than 88.02969.
FIX	Returns the truncated integer part of a numeric expression.
INT	Returns the largest integer value that is less than or equal to a given numeric expression.
LOG	Returns the natural logarithm of a numeric expression.
RANDOMIZE	Reseeds the random number generator.
RND	Returns a pseudo-random number between 0 and 1.
SGN	Returns 1 if a numeric expression is positive, returns 0 if the expression is equal to zero, and returns -1 if the expression is negative.
SIN	Returns the sine of a numeric expression which you must give in radians.
SQR	Returns the square root of a numeric expression.
TAN	Returns the tangent of a numeric expression which you must give in radians.

Derived Functions

BASIC provides intrinsic functions for your immediate use. From these intrinsic functions, you may derive the following functions:

Function	Equivalent
Secant	`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(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708`
Inverse Cosecant	`ARCCSC(X) = ATN(X/SQR(X*X-1)) + (SGN(X)-1) * 1.5708`
Inverse Cotangent	`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)+EXP(-X))`
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)-EXP(-X))`
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`

π/2

String Functions

The string functions manipulate string expressions.

ASC	Returns a numeric value that is the ASCII code of the first character of a string expression.
CHR$	Returns the character that corresponds to a given ASCII code.
HEX$	Returns a string expression that represents a hexadecimal value for a decimal argument.
INSTR	Searches for the first occurrence of a substring and returns the position where the match is found.
LEFT$	Returns a string expression comprised of the requested, leftmost characters of a string expression.
LEN	Returns the number of characters in a string expression.
MID$	Returns a substring from a given string expression.
OCT$	Returns a string that represents the octal value of a decimal argument.
RIGHT$	Returns a string expression comprised of the requested, rightmost characters in a string expression.
SPACE$	Returns a string of spaces the length of a numeric expression.
STR$	Returns a string representation of the value for a numeric expression.
STRING$	Returns a given length string whose characters all have the same ASCII code.
VAL	Returns the numeric value of a string expression.

Special Functions

BASIC provides the following special functions:

`ERR` and `ERL`	Direct program flow in an error-trap routine.
`FRE`	Forces "garbage collection".
`PEEK`	Returns the byte (decimal integer in the range 0 (eight zeros) to 255 (eight ones)) read from a memory location.
`USR`	Calls an assembly-language subroutine.
`VARPTR`	Returns the address of the first byte of data identified by a variable's name.

Chapter 6

BASIC STATEMENTS, COMMANDS, FUNCTIONS, AND VARIABLES

Introduction

This chapter contains a comprehensive listing of the commands, statements, functions, and variables that BASIC provides.

The distinction between commands and statements is mainly traditional. In general, commands operate on programs, and you usually enter them in Direct Mode. Statements direct the flow of control within a BASIC program.

Functions are predefined operations that perform a specific task. They return a numeric or string value. You can put the built-in functions and variables to immediate use.

Chapter Format

The statement and command descriptions take the following form:

Format: Shows the correct syntax for that instruction.

Purpose: Describes the instruction and what it does.

Remarks: Provides details on the instruction's use and supplies pertinent notes and comments.

Example: Gives an example of the instruction's use.

Since most of the functions perform familiar operations (such as taking the square root of a number or returning the sine of an angle), the chapter simplifies their treatment. Each description contains the function's format, its action, and an example:

Format: Shows the correct syntax for the function.

Action: Describes what the function does.

Example: Shows sample program segments that demonstrate the function's use.

ABS Function

Format: `ABS(x)`

Action: Returns the absolute value of the expression *x*.

Example:

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

ASC Function

Format: `ASC(x$)`

Action: Returns a numeric value that is the ASCII code of the first character in the string *x$*. (Appendix C lists the ASCII codes.)

If *x$* is the null string, an `Illegal function call` occurs.

See the `CHR$` function for ASCII-to-string conversions.

Example:

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

ATN Function

Format: `ATN(x)`

Action: Returns the arctangent of *x*, where *x* is given in radians. The result is in radians and ranges between -pi/2 and pi/2. The expression *x* may be any numeric type, but BASIC evaluates `ATN` in single-precision arithmetic.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.249046
Ok
```

AUTO Command

Format: AUTO [*line#* [, *increment*]]

Purpose: Generates a line number automatically when you press the [Return] key. You normally use this command when you are entering a program to free yourself from typing each line number.

Remarks: AUTO begins numbering at *line#* and increments each subsequent line number by *increment*. The default setting for both values is 10. If you follow *line#* with a comma but omit the increment, BASIC uses the increment specified in the last AUTO command.

When the AUTO command generates a line number that is already being used, BASIC prints an asterisk after the number to warn you that any characters you type will replace the existing line. If this is not your intent, you may press the [Return] key to preserve the old line and generate the next line number.

NOTE

Pressing the [Return] key must be your first action after the warning asterisk appears. If you happened to press a character before pressing the [Return] key, BASIC would replace the current line with that character.

Simultaneously pressing [CTRL] [C] stops the automatic generation of line numbers. Since pressing the [Return] key to end a line generates a new number for the next line, BASIC discards the line in which you press [CTRL] [C]. However, when the line in which you type [CTRL] [C] has an asterisk after the line number (showing that the line currently exists), BASIC preserves the line. BASIC returns control to the command level.

Examples: This first example generates line numbers beginning at 10 and incrementing by 10. (Ten is the default value for both the starting line number and the increment.):

```
AUTO
```

The next example generates the line numbers 100, 150, 200, etc.:

```
AUTO 100, 50
```

The last example generates line numbers beginning with 1000 and increasing by 50 at each step. (This example assumes that the next command follows the preceding command where the increment was 50.):

```
AUTO 1000,
```

NOTE

The BASIC compiler offers no support for this command.

BLOAD Command/Statement

Format: `BLOAD` *filename* [`,`*offset*]

Purpose: Loads the specified memory image file from disc into your computer's memory.

Remarks: *filename* is a string expression that contains the filename and an optional device designation. The filename portion may be 1 to 8 characters long.

When you omit the device designation in *filename*, BASIC assumes you are referring to the current drive.

offset is a numeric expression that returns an unsigned integer which may range between 0 and 65535. This is used in conjunction with a `DEF SEG` statement to specify an alternate location where loading begins.

As a command, you can use `BLOAD` to load assembly-language routines immediately into memory. A program can use `BLOAD` as a statement to selectively load assembly-language routines.

The `BLOAD` statement loads a program or data file (which you saved as a memory image file) anywhere in memory. A **memory image file** is a byte-for-byte copy of what was orginally in memory. For example, you may use `BLOAD` to load assembly-language programs, compiled Microsoft® Pascal programs, and Microsoft® FORTRAN routines. See the `BSAVE` command in this chapter for information about saving memory files.

When you omit the *offset* parameter, BASIC uses the segment address and offset that are contained in the file. (That is, the address you specified in the `BSAVE` statement when you created the file.) BASIC loads the file, therefore, back to the same location from which it was originally saved.

When you give an offset, BASIC uses the segment address from the most recently executed `DEF SEG` statement. Therefore, a program should execute a `DEF SEG` statement before it executes a `BLOAD` statement. If BASIC fails to encounter a `DEF SEG` statement, it uses the BASIC Data Segment (DS) as the default address.

CAUTION

Since BLOAD never performs an address range check, you may load a file anywhere in memory. You must be careful, therefore, to avoid loading a file over the BASIC interpreter program or the MS-DOS operating system.

Example: The following example sets the segment address at 6000 Hex and loads PROG1 at F000:

```
10 REM Load subroutine at 6F000
20 DEF SEG = &H6000 'Set segment to 6000 Hex
30 BLOAD "PROG1", &HF000 'Load PROG1
```

NOTE

The BASIC compiler offers no support for this command.

BSAVE Command/Statement

Format: BSAVE *filename*, *offset*, *length*

Purpose: Saves the contents of the specified area of memory as a disc file. (Also see the BLOAD statement.)

Remarks: *filename* is a string expression that contains the filename and an optional device designation. The filename portion may be 1 to 8 characters long.

offset is a numeric expression that returns an unsigned integer which may range between 0 and 65535. This is the offset address into the segment that you declared in the last DEF SEG statement. It specifies the exact location of the first byte of memory that is saved to disc.

length is a numeric expression that returns an unsigned integer which may range between 1 and 65535. This gives the length in bytes of the memory image file that you want to save.

The syntax for BSAVE requires all three parameters: *filename*, *offset*, and *length*. If you enter an improper *filename*, a Bad file name error occurs. Omitting *offset* or *length* produces a Syntax error. Under any of these circumstances, BASIC cancels the BSAVE operation.

Since the address given in the most recently executed DEF SEG statement determines the starting point from which BASIC calculates the offset, you should execute a DEF SEG statement before you execute a BSAVE statement.

Example: The following example saves 256 bytes, beginning at 6F000, in file PROG1:

```
10 REM SAVE PROG1
20 DEF SEG = &H6000
30 BSAVE "PROG1", &HF000, 256
```

NOTE

The BASIC compiler offers no support for this command.

CALL Statement (for Assembly Language Subroutines)

Format: CALL *varname* [(*argument* [,*argument*]. . .)]

Purpose: Calls an assembly-language subroutine.

Remarks: *varname* contains the segment offset that is the starting point in memory of the called subroutine. It cannot be an array variable name. You must assign the segment offset to the variable before you use the CALL statement.

argument is a variable or constant that is being passed to the subroutine. No literals are allowed. You must separate the items in the list with commas.

The CALL statement is the recommended way of calling machine-language programs with BASIC. You should avoid the USR function. See Appendix D, Assembly Language Subroutines.

The CALL statement generates the same calling sequence that is used by Microsoft® FORTRAN and Microsoft® BASIC compilers.

When the CALL statement executes, BASIC transfers control to the routine via the segment address given in the last DEF SEG statement and the segment offset specified by the *varname* parameter of the CALL statement. You may return values to the calling program by including within the list of arguments variable names to receive the results.

Example: This example sets the segment address to 8000 Hex. The variable FOO is set to &H7FA, so that the call to FOO executes the subroutine located at 8000:7FA Hex (equivalent to absolute address 807FA):

```
100 DEF SEG = &H8000
110 FOO = &H7FA
120 CALL FOO (A,B$,C)
```

NOTE

Refer to the BASIC compiler manual for differences between the interpretive and compiled versions of BASIC when using the CALL statement.

CALLS Statement

Format: `CALLS` *varname* [`(`*argument.list*`)`]

Purpose: Calls a subroutine with segmented addresses.

Remarks: The `CALLS` statement resembles the `CALL` statement, except the segmented addresses of all arguments are passed. A `CALL` statement passes unsegmented addresses. You should use the `CALLS` statement when accessing MS-FORTRAN subroutines, since all MS-FORTRAN parameters are call-by-reference segmented addresses.

As with the `CALL` statement, `CALLS` uses the segment address defined by the most recently executed `DEF SEG` statement to locate the routine being called.

NOTE

For more information, refer to Appendix D, "Assembly Language Subroutines".

CDBL Function

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
```

CHAIN Statement

Format: `CHAIN[MERGE]` *filename*`[ , [`*line*`][,ALL][,DELETE` *range*`]]`

Purpose: Calls a program and passes variables to it from the current program.

Remarks: *filename* is the name of the program that you are calling.

In the example:

```
CHAIN "PROG1"
```

BASIC searches the currently active disc for the file `PROG1.BAS`. When it locates the file, it loads then executes the program. Once the program resides in memory, you may list and modify it.

If BASIC fails to locate the file, it prints a `File not found` error message, and when no `ON ERROR` statement is active, halts execution and returns the user to command mode.

You may specify a different drive than the currently active one by including a letter specifer for the drive (followed by a colon) as part of *filename*. For example,

```
CHAIN "C:PROG2"
```

line is either a line number or an expression, which evaluates to a line number, in the called ("chained-to") program. It becomes the starting point for executing the called program. When you omit this parameter, BASIC begins executing the called program at the first line. The following statement begins executing `PROG1` at line 1000:

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

If BASIC fails to find the given line number, an `Undefined line number` error results.

Since *line* refers to a line in another program, a `RENUM` command has no effect on it. (`RENUM` only affects line numbers in the current (or calling) program.)

During the chaining process, the `CHAIN` statement leaves open any files that were opened.

The `ALL` option passes every variable in the current program to the called program. When you omit this parameter, the current program must contain a `COMMON` statement to list the variables that are being passed. An example of a `CHAIN` statement with the `ALL` option is:

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

The arguments for the `CHAIN` statement are position dependent. For example, when you use the `ALL` option but omit the starting line, you must include a comma to hold the place for the line parameter. That is, `CHAIN "NEXTPROG",,ALL` is correct while `CHAIN "NEXTPROG",ALL` is illegal. (In the latter statement, BASIC assumes `ALL` is a variable name for a line number expression.)

Including the `MERGE` option allows a subroutine to be brought into the BASIC program as an overlay. That is, BASIC merges the called program with the current program. The called program must be in ASCII format before you can merge it.

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

When using the `MERGE` option, you should place any user-defined functions before any `CHAIN MERGE` statements in that program. If they are not defined prior to the merge, they remain undefined after the merge operation is completed.

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

When you omit the `MERGE` option, the `CHAIN` statement does not preserve variable types or user-defined functions for use by the called program. That is, you must reissue any `DEFINT`, `DEFSNG`, `DEFDBL`, `DEFSTR`, or `DEFFN` statements within the called program.

After an overlay is brought in and finishes processing, you may delete it with the DELETE option. This allows BASIC to bring in a new overlay if one is needed.

```
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 number 1000.

NOTE

The CHAIN statement does a RESTORE before running the chained program. Therefore, the next READ statement accesses the first item in the first DATA statement that the program contains. The read operation does not continue from where it left off in the chaining program.

The RENUM command affects the line numbers in *range* since they refer to lines in the current program.

Example 1:

```
 5 REM -----------THIS IS PROGRAM 1 -----------
10 REM THIS EXAMPLE PASSES VARIABLES
15 REM USING THE "COMMON" STATEMENT
20 REM SAVE THIS MODULE ON DISK AS "PROG1" USING
       THE A OPTION
30 DIM A$(2), B$(2)
40 COMMON A$(), B$()
50 A$(1) = "VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2) = "VALUES BEFORE CHAINING."
70 B$(1) = " " : B$(2) = " "
80 CHAIN "PROG2"
90 PRINT : PRINT B$(1) : PRINT B$(2) : PRINT
100 END

 5 REM -----------THIS IS PROGRAM 2 -----------
10 REM STATEMENT 30 ABOVE "DIM A$(2), B$(2)"
       MAY ONLY BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30 REM SAVE THIS MODULE ON THE DISC AS "PROG2"
       USING THE A OPTION.
40 COMMON A$(), B$()
50 PRINT : PRINT A$(1); A$(2)
60 B$(1) = "NOTE HOW THE OPTION OF SPECIFYING A
             STARTING LINE NUMBER"
70 B$(2) = "WHEN CHAINING AVOIDS THE DIMENSION
             STATEMENT IN 'PROG1'."
80 CHAIN "PROG1",90
90 END
RUN "PROG1" [Return]
VARIABLES IN COMMON MUST BE ASSIGNED VALUES
BEFORE CHAINING.
NOTE HOW THE OPTION OF SPECIFYING A STARTING
LINE NUMBER WHEN CHAINING AVOIDS THE
DIMENSION STATEMENT IN 'PROG1'.
Ok
```

Example 2:

```
 5 REM -----------MAINPRG -------------------
10 REM THIS EXAMPLE USES THE MERGE, ALL, AND
       DELETE OPTIONS.
20 REM SAVE THIS MODULE ON THE DISC AS "MAINPRG".
30 A$ = "MAINPRG"
40 CHAIN MERGE "OVRLAY1",1010,ALL
50 END

1000 REM SAVE THIS MODULE ON DISC AS "OVRLAY1"
         USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO OVRLAY1."
1020 A$ = "OVRLAY1"
1030 B$ = "OVRLAY2"
1040 CHAIN MERGE "OVRLAY2", 1010, ALL,
          DELETE 1000-1050
1050 END

1000 REM SAVE THIS MODULE ON DISC AS "OVRLAY2"
         USING THE A OPTION.
1010 PRINT A$; " HAS CHAINED TO "; B$; "."
RUN "MAINPRG" [Return]
MAINPRG HAS CHAINED TO OVRLAY1.
OVRLAY1 HAS CHAINED TO OVRLAY2.
Ok
```

NOTE

The BASIC compiler offers no support for the ALL, MERGE, and DELETE options to the CHAIN statement. If you want to maintain compatibility with the BASIC compiler, you should pass variables with the COMMON statement and avoid using overlays.

CHR$ Function

Format: `CHR$(i)`

Action: Returns the character that corresponds to a given ASCII code.

You normally use `CHR$` to send special characters to the computer. For example, you could send the BELL character (`CHR$(7)`) as a preface to an error message.

See the `ASC` function for ASCII-to-numeric conversions.

Examples:

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

The following `PRINT` statement uses escape sequences to home the cursor and clear the display:

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

CINT Function

Format: `CINT(x)`

Action: Converts *x* to an integer by rounding off the fractional part.

x must be within the range of -32768 to 32767. If *x* is outside this range, an `Overflow` error occurs.

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

Example:

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

CLEAR Statement

Format: `CLEAR` [`,` [*expression1*] [`,` *expression2*]]

Purpose: Sets all numeric variables to zero and all string variables to the null string, closes all files, and, optionally, sets the end of memory and the amount of stack space.

Remarks: *expression1* sets the maximum number of bytes for the BASIC workspace. When you omit this parameter, BASIC uses all available memory up to the starting point of the MS-DOS operating system.

expression2 sets aside stack space for BASIC. When you omit this parameter, BASIC sets aside either 512 bytes or one-eighth of the available memory, whichever is smaller.

The `CLEAR` statement performs the following functions:

- Frees all memory used for data without erasing the program currently in memory
- Closes all files
- Clears all `COMMON` and user variables
- Resets the stack and string space
- Releases all disc buffers
- Resets all numeric variables and arrays to zero
- Resets all string variables and arrays to null
- Clears definitions set by any `DEF` statements. (This includes `DEF FN`, `DEF SEG`, and `DEF USR`, as well as `DEFINT`, `DEFSNG`, `DEFDBL`, and `DEFSTR`.)

Examples: The first example clears all data from memory without erasing the program:

```
CLEAR
```

The next statement clears all data and sets the maximum workspace size to 32K bytes:

```
CLEAR,32768
```

The next example clears all data and sets the size of the stack to 2000 bytes:

```
CLEAR,,2000
```

The last example clears all data and sets the maximum workspace size to 32K bytes and the stack size to 2000 bytes:

```
CLEAR,32768,2000
```

NOTE

If you intend to compile your program, consult the BASIC compiler manual for differences in implementation between the compiled and interpretive version of this command.

CLOSE Statement

Format: `CLOSE` [[#] *filenum* [,[#] *filenum. . .*]]

Purpose: Concludes input and output to a disc file.

Remarks: *filenum* is the number you gave the file when you opened it. A `CLOSE` statement with no arguments closes all open files and devices.

The association between a particular file and its file number ceases when the file is closed. Therefore, you may then reopen the file using the same or a different file number. Similarly, you may use the freed file number to open a new file.

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

The following instructions close all disc files automatically:

- `END`
- `NEW`
- `RESET`
- `RUN` without the `R` option
- `SYSTEM`

The `STOP` statement, however, never closes any disc files.

Example:

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

COMMON Statement

Format: `COMMON` *variable* [`,` *variable*] . . .

Purpose: Passes variables to a chained program.

Remarks: *variable* is the name of the passed variable. You specify array variables by appending a pair of parentheses "`()`" to the variable's name.

The BASIC interpreter accepts the number of dimensions for an array as in:

```
COMMON EMPLOYEE(3)
```

but treats it as equivalent to:

```
COMMON EMPLOYEE()
```

Also, the number in parentheses is the number of dimensions, not the dimensions themselves. For example, `EMPLOYEE(3)` could correspond to either of the following `DIM` statements:

```
DIM EMPLOYEE(20,4,2)
```

or

```
DIM EMPLOYEE(10,5,12)
```

You use the `COMMON` statement in conjunction with the `CHAIN` statement. You pass variables in the main program to variables in the chained program by listing each variable name in a `COMMON` statement.

Although `COMMON` statements may appear anywhere within a program, good programming practice dictates grouping them at the program's beginning.

You cannot name the same variable in multiple `COMMON` statements.

When you want to pass all the variables within a program, you should use the `CHAIN` statement with the `ALL` option and omit the `COMMON` statement.

Example: (Listing for FILE2)

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

(Listing for FILE1)

```
10 A = 10 : CUST$ = "MADELAIN" : B = 20
20 COMMON A, CUST$, B
30 CHAIN "FILE2"
RUN
MADELAIN          10               0
Ok
```

Notice in the above example that BASIC prints the value for the variable F(1) as 0. Since the COMMON statement for FILE1 omitted the array variable F, BASIC assigns a value of zero to F(1).

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the compile and interpretive versions of this statement.

CONT Command

Format: `CONT`

Purpose: Continues program execution after execution was suspended by either your typing [CTRL] [C] or the program encountering a `STOP` or `END` statement.

Remarks: You enter this command in Direct Mode.

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an `INPUT` statement, execution continues by reprinting the prompt (`?` or *prompt string*).

You normally use the `CONT` statement in conjunction with the `STOP` statement to debug a program. After execution stops, you may examine intermediate values by using Direct Mode statements. You may resume execution with the `CONT` statement (which continues with the next executable statement) or the Direct Mode `GOTO` statement (which continues execution at the specified line number).

You may also use `CONT` to resume execution after BASIC suspends execution upon its detecting an error condition. However, you may not use `CONT` to resume execution if you have modified the program (through edit commands) during the break.

Example: The following program and interactive session illustrates how you might use the `CONT` statement:

```
10 INPUT "ENTER PRICE", AMOUNT
20 IF AMOUNT < 20! THEN SURCHG=1!
30 STOP
40 TOTAL = AMOUNT + SURCHG
50 PRINT TOTAL
RUN
ENTER PRICE
```

(you type `15` [Return])

```
Break in 30
Ok
```

(you type `PRINT SURCHG` [Return])

```
 1
Ok
```

(you type `CONT` [Return])

```
 16
Ok
```

For more information, see the `STOP` statement.

NOTE

The BASIC compiler offers no support for this command.

COS Function

Format: `COS(x)`

Action: Returns the cosine of *x*, where *x* is given in radians.

To convert degrees to radians, multiply the angle by PI/180, where PI = 3.141593.

BASIC evaluates `COS` in single-precision arithmetic.

Example:

```
10 X = 2 * COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

CSNG Function

Format: `CSNG(x)`

Action: Converts *x* to a single-precision number.

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

Example:

```
10 A# = 975.342124#
20 PRINT A#; CSNG(A#)
RUN
 975.342124      975.3421
Ok
```

CVI, CVS, CVD Functions

Format: `CVI(`*2-byte string*`)`
`CVS(`*4-byte string*`)`
`CVD(`*8-byte string*`)`

Action: Converts string values to numeric values.

Random-access disc files store numeric values as strings. Therefore, when you read values from a random disc file, you must convert the strings 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.

See also `MKI$`, `MKS$`, `MKD$`.

Example:

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

DATA Statement

Format: `DATA` *constant* [`,`*constant*]

Purpose: Stores information (that is, numeric or string constants) for later access by a program's `READ` statements.

Remarks: *constant* may be a numeric or string constant.

Numeric constants may assume either an integer, fixed-point, or floating-point format. Numeric expressions are illegal.

You must place quotation marks around a string constant only if the string contains embedded commas or colons, or if it has significant leading or trailing spaces. Otherwise, you may omit the quotation marks.

`DATA` statements are nonexecutable. You may place them anywhere within the program.

A `DATA` statement may contain as many constants as you may fit on the input line. (You must separate the `DATA` items by commas or spaces.) A program's `READ` statements access the `DATA` statements in sequential order (by line number). Therefore, you may envision the data to be a continuous list of items, regardless of how many items are on a line or where the lines occur within the program.

The variable type given in the `READ` statement must agree with the corresponding constant in the `DATA` statement or a `Type mismatch` error occurs.

You may reread the information stored in a `DATA` statement by using the `RESTORE` statement.

Example:

```
10 DATA 80, 90, tonight, " dinner", 25
20 FOR I = 1 TO 5
30     READ A$
40     PRINT A$; " ";
50 NEXT
60 END
RUN
80 90 tonight  dinner 25
Ok
```

DATE$ Function

Format: `DATE$`

Action: Retrieves the current date.

The `DATE$` function fetches the date, which BASIC derives from the date set with the `DATE$` statement.

The `DATE$` function returns a 10-character string in the form:

mm-dd-yyyy

where:

mm is the month of the year. Values range from 01 to 12.

dd is the day of the month. Values range from 01 to 31.

yyyy is the year. Values range from 1980 to 2099.

Example:

```
PRINT DATE$
02-27-1984
Ok
```

DATE$ Statement

Format: `DATE$` = *string*

Purpose: Sets the current date for use by the `DATE$` function.

Remarks: *string* represents the current date. You may enter it in one of the following forms:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

where:

mm is the month of the year. Values range from 01 to 12.

dd is the day of the month. Values range from 01 to 31.

yy or *yyyy* is the year. Values range from 1980 to 2099. When you include only two digits, BASIC assumes 19 for the first two digits.

Example: This example demonstrates both forms for the year entry:

```
DATE$ = "01-01-1984"
Ok
PRINT DATE$
01-01-1984
Ok
DATE$ = "02-27-84"
Ok
PRINT DATE$
02-27-1984
Ok
```

DEF FN Statement

Format: `DEF FN` *name* [`(`*parameter* [`,`*parameter*] . . . `)`] *definition*

Purpose: Names and defines a function which the user writes.

Remarks: *name* must be a legal variable name. This name, preceded by the letters `FN`, becomes the name of the function.

parameter is a variable name in the function definition that BASIC replaces with a value when the function is called. You must separate multiple parameters with commas.

definition is an expression that performs the operation of the function. You must limit the definition to one line (255 characters). Variable names that appear in this expression serve only as formal parameters to define the function. They have no effect on program variables that have the same name. A variable name used within the function definition might appear as a *parameter*. If it is a parameter, BASIC supplies its value when the function is called. Otherwise, BASIC uses the variable's current value.

The parameter variables correspond on a one-to-one basis to the argument variables or values that are given in the function call.

User-defined functions may be numeric or string. When the function name contains a type definition character, the value of the expression is forced to that type before BASIC returns the result to the calling statement. When you omit the type definition character, BASIC considers the result to be a single-precision value. When a type is specified in the function name and the argument type differs, 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.

The `DEF FN` statement is illegal when you are using the BASIC interpreter in Direct Mode.

Example 1: If a program contains the following lines:

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

Then defining a function such as:

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

simplifies the program to:

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

Example 2:

```
10 DEF FNMULT(I,J) = I*J+(I^2)*J+(I^3)*J
20 I = 2 : J = 3
30 A = FNMULT(I,J)
40 B = FNMULT(3,4)
50 PRINT A, B
RUN
 42      156
Ok
```

DEF SEG Statement

Format: `DEF SEG [=address]`

Purpose: Assigns the current "segment" for storage. A subsequent `BLOAD`, `BSAVE`, `CALL`, `CALLS`, `POKE`, `PEEK`, or `USR` instruction defines the actual physical address that it requires as an offset into this segment.

Remarks: *address* is a numeric expression that returns an unsigned integer which may range between 0 and 65535.

Entering an address outside the permissible range results in an `Illegal function call`. Under these circumstances, any previous value remains in effect.

BASIC saves the address you specify for use as the segment needed by a `BLOAD`, `BSAVE`, `CALL`, `CALLS`, `POKE`, `PEEK`, or `USR` instruction.

When you give an address, you should ensure that it is based on a 16-byte boundary. The value is multiplied by 16 (shifted left by 4 bits) to form the segment address for the subsequent operation. BASIC does not check the validity of the specified address.

When you omit the *address* parameter, BASIC sets the segment address to that of the BASIC Data Segment (DS). This is the setting for the current segment when you initialize BASIC.

NOTE

You must separate `DEF` and `SEG` with a space. Otherwise, BASIC interprets the statement:

```
DEFSEG = 1000
```

as "assign the value of 1000 to the variable `DEFSEG`".

Example: This example sets the segment address to &HB800 Hex. Later, a second statement (with no specified address) restores the address to the BASIC Data Segment (DS):

```
10 DEF SEG = &HB800
. . .
90 DEF SEG
```

DEF USR Statement

Format: `DEF USR` [*digit*] = *offset*

Purpose: Gives the starting address of an assembly-language subroutine.

Remarks: *digit* may be any integer from 0 to 9. The digit corresponds to the number of the `USR` routine that you are specifying. When you omit the *digit* parameter, BASIC assumes the reference is to `USR0`.

offset is an integer expression whose value may range from 0 to 65535. BASIC adds *offset* to the value of the current storage segment to get the actual starting address of the `USR` routine. (See Appendix D for information about assembly-language subroutines.)

`DEF USR` lets the programmer define starting addresses for user-defined assembly language functions that are called from BASIC programs. You must use this statement to set the starting address prior to its actual use.

A maximum of 10 user-defined functions are available for use at any given time. The routines are identified as `USR0` to `USR9`. When you need access to more subroutines, you can use multiple `DEF USR` statements to redefine a subroutine's starting address. However, BASIC only saves the last-executed value as the offset for that subroutine.

NOTE

The `CALL` statement is the preferred way of calling subroutines. You should avoid using the `USR` statement.

Example: This example calls the user function at the Data Segment relative memory location 24000:

```
200 DEF SEG = 0
210 DEF USR0 = 24000
220 X = USR0 (Y^2/2.89)
```

DEFINT/SNG/DBL/STR Statements

Format:

DEFINT *letter* [-*letter*] [,*letter* [-*letter*]]...
DEFSNG *letter* [-*letter*] [,*letter* [-*letter*]]...
DEFDBL *letter* [-*letter*] [,*letter* [-*letter*]]...
DEFSTR *letter* [-*letter*] [,*letter* [-*letter*]]...

Purpose: Declares that BASIC should automatically treat certain variable names as integer, single-precision, double-precision, or string variables, respectively.

Remarks: *letter* is a letter of the English alphabet (A-Z).

BASIC considers any variable names beginning with the specified letter(s) to be of the requested type. However, when assigning variable types, BASIC always gives precedence to a type declaration character (%,!,#, or $) over an assignment set by a DEF*type* statement.

In the following example, BASIC prints the variable C as an integer because of the type declaration character (%), even though C is within the range of the DEFDBL declaration.

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

When you use these statements, you should place them at the beginning of a program. (BASIC must execute the DEF*type* statement before you use any variables that it declares.)

If a program contains no type declaration statements, BASIC assumes that any variable without a declaration character is a single-precision variable.

Examples: The first example defines all variables that begin with either the letter L,M,N,O, or P to be double-precision variables:

```
10 DEFDBL L-P
```

The next statement defines all variables that begin with the letter A to be string variables:

```
10 DEFSTR A
```

The last example defines all variables that begin with either the letter I,J,K,L,M, N,W,X,Y, or Z to be integer variables:

```
10 DEFINT I-N,W-Z
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

DELETE Command

Format: `DELETE` [*start.line*] [`-`*end.line*]

Purpose: Deletes the specifed line(s) from a BASIC program.

Remarks: *start.line* is the line number for the first line you want to delete.

end.line is the line number for the last line you want to delete.

NOTE

You may use a period (`.`) in place of a line number when you want to delete the current line.

If BASIC fails to find the line number you supplied, it returns an `Illegal function call`.

BASIC always returns control to the command level after the `DELETE` command executes.

Examples: The first example only deletes line 40:

```
DELETE 40
```

The next statement deletes from the beginning of the program through line 40:

```
DELETE -40
```

The last example deletes all lines between 40 and 80, inclusively:

```
DELETE 40-80
```

NOTE

The BASIC compiler offers no support for this command.

DIM Statement

Format: `DIM` *arrayname* `(`*subscripts*`)` [`,`*arrayname* `(`*subscripts*`)`]. . .

Purpose: Sets the maximum values for the subscripts of an array variable, allocates the necessary storage, and initializes the elements of the array to zero or null.

Remarks: *arrayname* is a variable that names the array.

subscripts is a list of numeric expressions, separated by commas, that define the array's dimensions.

When you fail to dimension an array with the `DIM` statement, BASIC assumes the maximum subscript is 10. If you subsequently use a subscript that exceeds this number, a `Subscript out of range` error occurs.

If you attempt to dimension an array more than once, a `Duplicate Definition` error occurs. (See the `ERASE` statement.)

The minimum value for an array subscript is zero unless you use the `OPTION BASE` statement to change it to one.

The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. However, these values are theoretical limits as both values are limited by the size of memory and the number of characters that you can enter on the input line.

The `DIM` statement sets all elements of numeric arrays to an initial value of zero and all elements of string arrays to the null string.

Example:

```
10 DIM ID(20)
20 FOR I = 0 TO 20
30     READ ID(I)
40 NEXT I
```

NOTE

If you plan to compile your program, see the BASIC complier manual for differences between the compiled and interpretive version of the `DIM` statement.

EDIT Command

Format: `EDIT` *line*
`EDIT .`

Purpose: Enables Edit mode on the specified line.

Remarks: The `EDIT` command displays the specified line number, then waits for an Edit subcommand. You may then modify the line with any of the techniques presented in Chapter 1.

When you specify a line number, the `EDIT` command edits that line. If no such line exists, an `Undefined line number` error occurs.

When you enter `EDIT .`, the `EDIT` command edits the last line that you typed, the last line that a `LIST` statement displayed, or the last line that an error message referenced.

Examples: Both of the following groups of commands display Line 10 for editing:

```
EDIT 10
```

```
LIST 10
EDIT .
```

NOTE

The BASIC compiler offers no support for this command.

END Statement

Format: END

Purpose: Stops program execution, closes all files, and returns control to the command level.

Remarks: You may place END statements anywhere in a program to end execution. The END statement at the end of a program, however, is optional. When you omit it, execution stops after the last line in the program executes.

The END statement differs from the STOP statement in two important ways:

- END closes all files
- END terminates the program without printing a Break message

BASIC always returns control to the command level after an END statement executes.

Example: This program segment tests to see if more data exists. END statements terminate the program when no data exists and prevent program flow from falling into the subroutine section:

```
520 IF EOF(1) THEN END ELSE GOTO 200
. . .
850 END
1000 REM THE FOLLOWING SECTION CONTAINS
1010 REM THE INPUT SUBROUTINES
```

NOTE

If you plan to compile your program, refer to the BASIC Compiler Manual for programming differences when using the END statement.

EOF Function

Format: `EOF(`*filenum*`)`

Action: For sequential files, the `EOF` function returns true (-1) when no more data exists in the file. BASIC considers the file empty if the next input operation (for example, `INPUT` or `LINE INPUT`) would cause an `Input past end` error. Using the `EOF` function to test for the end-of-file while inputing information avoids such errors.

For random-access files, `EOF` returns true (-1) if the most recently executed `GET` statement attempts to read beyond the end-of-file.

Because BASIC allocates 128 bytes to a file at a time, it is possible that `EOF` will not accurately detect the end of a random-access file that was opened with a record length of less than 128 bytes. For example, if you open a file with a record length of 64 bytes and you write one record to the file (that is, `PUT #1,1`), `EOF` returns false if a `GET` statement is attempted on the file's record (for example, `GET #1,1`). This occurs even though the record has not actually been written.

Example: This sample program lists the titles of the books cataloged in the file `LIBRARY.DAT`. It also counts the books in the library by counting the number of records that it reads from `LIBRARY.DAT` before it encounters the end-of-file.

Each record of `LIBRARY.DAT` contains information on one book. The record length is 128 bytes. The first 35 bytes contain the title of the book. The remaining 93 bytes contain additional information such as the author, publisher, print date, and so on.

```
10 REM
20 REM  Open the library catalog file,
30 REM  LIBRARY.DAT.
40 OPEN "R",1,"LIBRARY.DAT"
50 REM  The first 35 bytes of the
60 REM  record contain the title,
70 REM  the remaining 93 bytes contain
80 REM  additional information that
90 REM  this program does not use.
100 FIELD 1, 35 AS TITLE$, 93 AS G$
110 REM
120 REM  Initialize the number of books seen.
130 REM
140 NBOOKS = 0
150 REM  Attempt to fetch the next record.
160 REM  Note that the record number
170 REM  of GET isn't specified
180 REM  so the next record of the file
190 REM  is fetched.
200 GET 1
210 REM
220 REM  Is this the end of the file?
230 REM
240 IF EOF(1) THEN 1000
250 REM  If no: increment the count of books,
260 REM  print the current title, and
270 REM  loop back to read the next record.
280 REM
290 NBOOKS = NBOOKS + 1
300 PRINT TITLE$
310 GOTO 200
1000 REM  Control passes here when the end of
1010 REM  file has been reached, so:
1020 REM  print a blank line and the number of
1030 REM  books, close the file, and terminate
1040 REM  the program.
1050 PRINT "There are "; NBOOKS; " books in ";
1060 PRINT "your library."
1070 CLOSE
1080 END
```

ERASE Statement

Format: `ERASE` *arrayname* [`,`*arrayname*] . . .

Purpose: Deletes the named arrays from the program.

Remarks: *arrayname* names the array that you want to delete.

After you delete an array, you may redimension that array or use the previously allocated array space for another purpose.

Attempting to redimension an array without first erasing it causes a `Duplicate Definition` error.

Example:

```
450 ERASE ID, STATS
460 DIM ID(99)
```

NOTE

The BASIC compiler offers no support for this statement.

ERR and ERL Variables

Format:

```
ERR
ERL
```

Action: When BASIC enters an error-handling routine, the variable `ERR` contains the error code for the error, and the variable `ERL` contains the line number of the line in which BASIC detected the error.

You normally use these variables in `IF...THEN` statements to direct program flow in the error trap routine.

When the statement causing the error was a Direct Mode statement, `ERL` contains the value 65535. To test if an error occurred in a Direct Mode statement requires the following statement:

```
IF ERL = 65535 THEN . . .
```

You may also test for other error conditions by using the following statements:

IF ERR = *error.code* THEN

or

IF ERL = *line#* THEN

You could also enter the previous statement as:

IF *line#* = ERL THEN

However, when *line#* appears on the left side of the equal sign, the `RENUM` command fails to adjust the value for *line#* if its value changes while resequencing the program.

CAUTION

Numeric constants following an `ERL` variable in a given expression may be treated as line references and thus modified by a `RENUM` statement. To avoid this problem, you should use statements similar to these:

```
L = ERL : PRINT L/10
```

rather than this statement:

```
PRINT ERL/10
```

`ERL` and `ERR` are variables that BASIC reserves for its use. Therefore, BASIC prevents you from assigning values to these variables. For example, the following assignment is illegal:

```
LET ERR = 65535
```

Appendix A lists the BASIC error codes.

ERROR Statement

Format: `ERROR` *number*

Purpose: Either simulates the occurrence of a BASIC error or allows you to define error codes.

Remarks: *number* must be an integer expression between 0 and 255. When the value of *number* is equal to a BASIC error message, the `ERROR` statement simulates the occurrence of that error (which includes the printing of the corresponding error message). (See Example 1.)

To define your own error code, select a value that is greater than those used by the BASIC error codes. (We recommend that you use the highest available values, for example numbers over 200, so your program can maintain compatability if BASIC adds more error codes in later version of this package.) This user-defined error code may then be conveniently handled in an error-trap routine. (See Example 2).

When an `ERROR` statement specifies a code for an error message that is undefined, BASIC responds with the message `Unprintable error`.

Executing an `ERROR` statement for which no error-trap routine exists prints an error message and halts execution.

Example 1:

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

If you are using the BASIC interpreter in Direct Mode, you may enter an error number at the `Ok` prompt.

For example, if you enter:

```
ERROR 15
```

BASIC responds:

```
String too long
Ok
```

Example 2:

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

EXP Function

Format: `EXP(x)`

Action: Returns *e* (where *e* = 2.71828. . .) to the power of *x*. The number *e* is the base of the natural logarithms.

x must be less than 88.02969.

If `EXP` overflows, BASIC displays the `Overflow` error message, sets the result to machine infinity with the appropriate sign, and continues execution.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.59815
Ok
```

FIELD Statement

Format: `FIELD` [#] *filenum* ,*field.width* `AS` *stringvar* [,*field.width* `AS` *stringvar*] . . .

Purpose: Allocates space for variables in the random file buffer.

Remarks: BASIC reads and writes random files through a file buffer that holds the file record. You must assemble and disassemble this buffer into individual variables. Therefore, this requires your using the `FIELD` statement to specify the layout of the file buffer before you get data out of a random file buffer after a `GET`, or to enter data before a `PUT`.

filenum is the number you gave the file when you opened it.

field.width is the number of character positions that you want to allocate to *stringvar*. For example, the following statement allocates the first 20 positions (bytes) in the random file buffer to the string variable `CNAMES$`, the next 10 bytes to `ID$`, and the next 40 bytes to `ADDRESS$`:

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

stringvar is a string variable that is used for random file access.

The `FIELD` statement is a template for formatting the random file buffer. It never places any data into the buffer. (See the `GET` and `LSET/RSET` statements for information on moving data into and out of the random file buffer.)

You may execute any number of `FIELD` statements for a given file. Once it executes, a `FIELD` statement remains in effect. Each new `FIELD` statement redefines the buffer from the first character position. This permits multiple field definitions for the same data.

The total number of bytes you allocate with a `FIELD` statement must not exceed the record length that you set when you opened the file. (When you omit specifying the length parameter, BASIC sets the record length to 128 bytes.) Attempting to allocate more bytes than the record can hold results in a `Field overflow` error.

If your definition of a record's layout requires more than 255 characters, you must divide the definition into two or more `FIELD` statements. For example:

```
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 ALTCODES, 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 DUMMY$, 40 AS COMMENTS$,
     18 AS FRSTNAME$
```

In this example, `DUMMY$` is a string variable whose width is equal to the combined width of all the variables in the previous `FIELD` statement. It provides a way of skipping over the buffer space that you allocated to variables in the first `FIELD` statement. Never assign a `LSET` or `RSET` value to these dummy variables.

NOTE

Be careful how you use a field variable name in an `INPUT` or `LET` statement. After you assign a variable name to a field, it points to the correct place in the random file buffer. If a subsequent `INPUT` or `LET` statement with that variable's name executes, the variable's pointer moves to string space and ceases to be in the file buffer.

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
```

FILES Command/Statement

Format: FILES [*filename*]

Purpose: Lists the names of the files that reside on the specified disc.

Remarks: *filename* is a string expression that contains the file's name and an optional device designation.

filename may contain question marks (?) or asterisks (*) as **wild cards**. A question mark matches any single character in the filename or extension. For example, CHAP? would match CHAP1, CHAP2, CHAPS, and so on. An asterisk matches one or more characters, beginning at that position. For example, CHAP* not only matches all the files listed above but also matches CHAPTER, CHAPLAIN, CHAPEAU, and so on.

Omitting *filename* lists all the files on the currently selected drive.

Examples: This statement lists all the files on the current disc:

```
FILES
```

The next statement lists all files with the BASIC file type extension (.BAS):

```
FILES "*.BAS"
```

This statement lists all the BASIC files with a PROG prefix and one trailing character, such as PROGS.BAS or PROG1.BAS:

```
FILES "PROG?.BAS"
```

The last statement lists all the files on the disc in drive B:

```
FILES "B:*.*"
```

FIX Function

Format: FIX(*x*)

Action: Returns the truncated integer portion 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*.
For example,
  FIX(-3.99) returns -3
whereas
  INT(-3.99) returns -4.

Examples:

```
PRINT FIX (58.75)
 58
Ok

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

FOR. . .NEXT Statement

Format: `FOR` *variable* `=` *x* `TO` *y* [`STEP` *z*]

. . .

[*loop statements*] . . .

`NEXT` [*variable*] [`,` *variable*] . . .

Purpose: Loops through a series of statements a given number of times.

Remarks: *variable* serves as a counter.

x, *y*, and *z* are numeric expressions.

x is the initial value of the counter.

y is the final value of the counter.

z is the increment. When you omit this parameter, BASIC increments the count by one on each iteration through the loop. If `STEP` is negative, the final value of the counter is set to be less than the initial value. Under these circumstances, BASIC decrements the counter on each iteration through the loop, and looping continues until the counter is less than the final value.

BASIC executes the program lines that follow the `FOR` statement until it encounters the `NEXT` statement. BASIC then increments the counter by the amount specified by `STEP`. It then checks to see if the value of the counter exceeds the final value (*y*). If it is not greater than the final value, BASIC branches back to the first statement within the loop and repeats the process. When the counter finally exceeds the final value, execution continues with the statement after the `NEXT` statement.

You may modify the value of *variable* from inside the loop. However, we do not recommend this practice.

If the initial value of the loop times the sign of the step exceeds the final value times the sign of the step, BASIC skips over the `FOR...NEXT` loop.

You may place a `FOR...NEXT` loop within the context of another `FOR...NEXT` loop. When you nest loops in this fashion, each loop must have a unique variable name for its counter. Furthermore, the NEXT statement for the inner loop must appear before the NEXT statement of the outer loop. When nested loops have the same end point, you may use a single NEXT statement for all of them.

The variable name(s) in the NEXT statement are optional.

If a NEXT statement is encountered before its corresponding FOR statement, BASIC displays a NEXT without FOR error and halts execution.

Examples: Although the following example modifies the loop's final value, it has no effect on program execution since BASIC calculates this value only once when it first enters the FOR statement:

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30     PRINT
40     FOR J = 1 TO 3
50         K = K + 1
60         PRINT K;
70     NEXT J
80 NEXT I
90 END
RUN
 11  12  13
 14  15  16
 17  18  19
 20  21  22
 23  24  25
Ok
```

BASIC skips the FOR loop in the following example since the inital value of the loop exceeds the final value and a negative STEP doesn't appear:

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

The loop in the next example executes ten times since BASIC always calculates the final value for the loop value before it sets the initial value.

NOTE

Previous versions of BASIC set the initial value of the loop variable before setting the final value. Were this still true in the following example, the loop would have executed 6 times and not 10.

```
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 the statement,

```
FOR I = 45 TO 45.8 STEP 0.2
```

BASIC executes the loop four times; and not five times as you would expect. This results from the computer's attempt to represent decimal digits in a binary format.

On each iteration of the loop, the value for the counter takes on these values:

45
45.20000076293945
45.40000152587891
45.60000228881836
45.80000305175781

As the last value exceeds 45.8, the `FOR` loop terminates after the fourth iteration.

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive versions of this statement.

FRE Function

Format: `FRE(0)`

`FRE(x$)`

`FRE("")`

Action: Returns the number of bytes of memory that are available for the user's program.

The `FRE` arguments are dummy arguments.

`FRE("")` forces the system to reorganize the memory that BASIC uses, so no space is used by unreferenced variables. It then returns the number of free bytes. BE PATIENT: this process can take from one to two minutes.

BASIC does not initiate memory consolidation until it uses its allotment of free memory. Therefore, using `FRE("")` periodically results in shorter delays for each memory reorganization.

Example:

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

GET Statement

Format: `GET` [#] *filenum* [, *recnum*]

Purpose: Reads a record from a random disc file into the random file buffer.

Remarks: *filenum* is the number you gave the file when you opened it.

recnum identifies the record to be read. The value for *recnum* may range from 1 to 32767.

When you omit *recnum,* BASIC reads the next record, which followed the last `GET`, into the buffer.

NOTE

After a `GET` statement, you may use the `INPUT#` statement and/or the `LINE INPUT#` statement to read characters from the random file buffer.

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 IF CODE% = 0 THEN END
50 GET #1, CODE%
60 PRINT CODE%
70 PRINT USING "$$###.##"; CVS(PRICE$)
80 PRINT CITY$ : PRINT
90 GOTO 30
```

GOSUB. . .RETURN Statement

Format: `GOSUB` *line#*

. . .

`RETURN`

Purpose: Branches to and returns from a subroutine.

Remarks: *line#* is the first line of the subroutine.

Subroutines allow you to key in a group of statements once, yet access them from different parts of a program. The `GOSUB` statement directs program flow to a subroutine, and sets up the mechanism to return control to the line following the `GOSUB` statement when the subroutine finishes execution.

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.

A subroutine's `RETURN` statement causes BASIC to branch back to the statement following the most recently executed `GOSUB` statement. A subroutine may contain more than one `RETURN` statement when program logic dictates returning from different parts of the subroutine.

Although subroutines may appear anywhere within a program, good programming practice recommends that subroutines be readily distinguishable from the main program. You may precede a subroutine with a `STOP`, `END`, or `GOTO` statement to direct program control around the subroutine. (This prevents program control from inadvertantly "falling through" a subroutine.)

Example:

```
10 PRINT "MAIN PROGRAM"
20 GOSUB 60
30 PRINT "BACK FROM SUBROUTINE"
40 END
50 REM ***** SUBROUTINE SECTION *****
60 PRINT "SUBROUTINE ";
70 PRINT "IN ";
80 PRINT "PROGRESS"
90 RETURN
RUN
MAIN PROGRAM
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

The END statement in line 40 prevents the subroutine from being executed a second time.

GOTO Statement

Format: `GOTO` *line*

Purpose: Branches directly to the specified line number.

Remarks: *line* is the line number of a statement in the program.

When *line* is an executable statement, BASIC executes that statement and program flow continues from there. When *line* is a nonexecutable statement (such as `REM` or `DATA`), execution continues at the first executable statement following *line*.

In Direct Mode, you may use the `GOTO` statement to reenter a program at a desired point. This can aid debugging.

Example: Indirect Mode

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

Direct Mode

```
GOTO 20
RADIUS = 12       AREA = 452.16
Out of DATA in 10
Ok
```

NOTE

You may use the `GOTO` statement in Direct Mode. However, if you precede this command with any other command that might change the values of variables (such as `CLEAR` or `RESTORE`), your results will differ.

HEX$ Function

Format: `HEX$(x)`

Action: Returns a string that represents the hexadecimal value of the decimal argument.

BASIC rounds *x* to an integer before it evaluates `HEX$(X)`.
See the `OCT$` function for octal conversions.

Example:

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

IF Statement

Format 1: IF *expression* [,] THEN {*clause* | [GOTO] *line*} [ELSE {*clause* | *line*}]

Format 2: IF *expression* GOTO *line* [ELSE {*clause* | *line*}]

Purpose: Determines program control based upon the result of the logical expression.

Remarks: *expression* is any logical (numeric) expression.

clause is either a BASIC statement or a sequence of statements that you separate with colons (:).

line is the line number of a statement in the program.

When the result of the *expression* is true (not zero), BASIC executes the THEN or GOTO clause. Consider this example:

```
10 INPUT I
20 PRINT I
30 IF I THEN GOTO 50
40 STOP
50 PRINT "HI!"
60 END
RUN
? 1 [Return]
 1
HI!
```

When *expression* is false (zero), BASIC disregards the THEN or GOTO clause and executes the ELSE clause if it is present. Otherwise, execution continues with the next executable statement. Consider this example:

```
10 INPUT I
20 PRINT I
30 IF I THEN GOTO 50
40 STOP
50 PRINT "HI!"
60 END
RUN
? 0 [Return]
 0
Break in 40
Ok
CONT [Return]
HI!
Ok
```

You may follow the reserved word THEN with either a line number where program control should branch, or with one or more statements to be executed.

You may place a comma before THEN.

You can only use a line number after the reserved word GOTO.

Nesting of IF Statements:

You may nest IF...THEN...ELSE statements to any depth, limited only by the length of the input line (255 characters). For example, the following statement is legal:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X THEN PRINT
"LESS THAN" ELSE PRINT "EQUIVALENT"
```

When an IF statement contains a different number of ELSE and THEN clauses, BASIC pairs each ELSE with the closest unmatched THEN. In the following example, the single ELSE clause pairs with the second THEN; not the first.

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

When you are conversing with the BASIC interpreter in Direct Mode and if you follow an IF...THEN statement with a line number, the interpreter displays an Undefined line number error message unless you have previously entered that line while in Indirect Mode.

NOTE

When using the IF statement to test equality for a value that results from a floating point computation, you should remember that the internal representation of the value is not exact. (This happens because a decimal number is being represented in binary format.) Therefore, you should conduct the test against the range of values over which accuracy 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. . .
```

The recommended method returns true if the value of A is between .999999 and 1.000001 (a relative error of less than 1.0E-6).

Examples: This statement gets record number I if I is not zero:

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

The following program segment tests whether I is between 10 and 20. If I is within this range, BASIC calculates a value for DB and branches to line 300. If I is outside this range, execution continues with line 110:

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

The next example selects a destination for printed output, depending on the value of a variable (IOFLAG). If IOFLAG is zero (false), output goes to the line printer; otherwise, output goes to the computer screen:

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

INKEY$ Function

Format: `INKEY$`

Action: Returns a one-character string that contains a character read from the computer's keyboard or the null string when no character is pending. `INKEY$` suppresses the echoing of the character to the screen.

Control-C terminates the program. All other characters are passed directly to the program.

Example:

```
10 PRINT "PRESS A KEY"
20 A$ = INKEY$
30 IF A$ ="" THEN GOTO 20
40 PRINT "YOU PRESSED THE "; A$; " KEY"
50 END
```

INP Function

Format: `INP(`*j*`)`

Action: Returns the byte read from the input port *j*. *j* may range from 0 to 65535.

NOTE

The input port is a microprocessor port. It does not refer to your computer's datacomm (or peripheral) ports.

`INP` is the complementary function to the `OUT` statement.

Example:

```
100 A = INP(2)
```

INPUT Statement

Format: `INPUT` [;] ["*prompt*" {; | ,} *variable* [, *variable*] . . .

Purpose: Takes input from the keyboard during program execution. BASIC accepts the data after you press the [Return] key.

Remarks: *prompt* is a string constant that assists the user in entering the proper information.

variable is the name of the numeric or string variable that receives the input. The variable may be a simple variable or the element of an array.

When BASIC encounters an `INPUT` statement, it prints a question mark (`?`) to show that the program is waiting for data. When you include *prompt*, BASIC displays that string before the question mark. You may then enter the requested data from the keyboard.

You may use a comma (`,`) instead of a semicolon after the prompt string to suppress the question mark. For example, the following statement prints the prompt without the trailing question mark:

```
INPUT "ENTER BIRTHDATE ", BDAY$
```

When you place a semicolon immediately after the reserved word `INPUT`, pressing the [Return] key does not echo a carriage return/line feed sequence:

```
10 PRINT "FOR EXAMPLE"
20 INPUT; A$
30 INPUT; B$
RUN
FOR EXAMPLE
? A [Return]? B [Return]
Ok
```

As you enter the necessary data, BASIC assigns the values to the listed variable(s). You must separate a series of items with commas, and the number of items you enter must agree with the number of variables in the list.

Responding to a prompt with too many or too few items, or the wrong type of value (string instead of numeric, for instance), prints the message `?Redo from start`. BASIC makes no assignment of values until it receives a completely acceptable response. For example,

```
10 INPUT "ALPHA PLEASE :", A$
20 INPUT "NUMBER ONLY :", B
30 PRINT "********A$=", A$
40 PRINT "********B =", B
RUN
ALPHA PLEASE :ALFA
NUMBER ONLY  :24
********A$=   ALFA
********B =    24
Ok
RUN
ALPHA PLEASE :BETA
NUMBER ONLY  :B
?Redo from start
NUMBER ONLY  :48
********A$=   BETA
********B =    48
Ok
```

When entering string information to an `INPUT` statement, you may omit surrounding the text with quotation marks.

If the prompt requests a single respond, you may press the [Return] key to enter a zero for a numeric item or the null string for a string variable.

Example 1:

```
10 INPUT X
20 PRINT X " SQUARED IS " X^2
30 END
RUN
```

`?` (you type `5` [Return])

```
5 SQUARED IS 25
Ok
```

Example 2:

```
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 END
RUN
WHAT IS THE RADIUS?
```

(you type 7.4 [Return])

```
THE AREA OF THE CIRCLE IS 171.9464
Ok
```

Example 3:

```
10 INPUT "ENTER THREE VALUES: ", A,B,C
20 AVE = (A+B+C)/3
30 PRINT "THE AVERAGE IS "; AVE
RUN
ENTER THREE VALUES:
```

(you type: 5,10,9 [Return])

```
THE AVERAGE IS 8
Ok
```

Example 4:

```
10 INPUT; "ENTER EMPLOYEE NUMBER"; ID
20 IF ID<25 THEN PRINT " INCORRECT VALUE"
. . .
RUN
```

(you type 5 [Return] to the prompt)

```
ENTER EMPLOYEE NUMBER? 5 INCORRECT VALUE
```

INPUT# Statement

Format: `INPUT#` *filenum*, *variable* [, *variable*] . . .

Purpose: Reads data values from a sequential disc file and assigns them to program variables.

Remarks: *filenum* is the number you gave the file when you opened it for input.

variable is the name of a numeric or string variable that receives the value read from the file. The variable may be a simple variable or an array element.

The `INPUT#` statement suppresses printing of the question mark as a prompt character.

Data items in a file should appear exactly as they would if you were typing the information as a response to an `INPUT` statement.

The items read must match the variable type of each variable.

For numeric values, BASIC discards any leading spaces, carriage return characters, or line feed characters. The first character that BASIC encouters that is not a space, carriage return, or line feed character is taken to be the beginning of a number. The number terminates on a space, comma, carriage return, or line feed character.

When BASIC scans a sequential file for a string value, it also discards any leading spaces, carriage returns, or line feed characters. The first character that it encounters that is not one of these three characters is taken to be the start of a string item. When the first character is a quotation mark ("), the string consists of all characters that occur between the first quotation mark and the second. Thus a quoted string cannot contain embedded quotation marks. When the first character is not a quotation mark, BASIC considers the string to be unquoted. In this case, the string terminates on a comma, carriage return, or line feed, or after 255 characters have been read.

If BASIC reaches the end-of-file while reading a numeric or string value, it terminates the item immediately.

Example:

```
10 OPEN "I", #1, "BUDG"
20 INPUT #1, CHCKNUM$, PAYEE$
30 PRINT CHCKNUM$, PAYEE$
40 GOTO 20
RUN
2134                ELECTRIC COMPANY
2136                GAS BILL
```

INPUT$ Function

Format: `INPUT$ (`*i* [`,`[`#`] *j*]`)`

Action: Returns a string of *i* characters.

i is the number of characters to be read from the file.

j is the file number that you used to open a file. Including the *j* parameter reads the string from that file. If you omit the *j* parameter, `INPUT$` reads the string from the computer's keyboard. When the keyboard serves as the source of input, `INPUT$` suppresses the echoing of characters to the screen and passes through all characters including control characters. The only exception is Control-C, which you may use to interrupt the execution of the `INPUT$` function and return control to the BASIC command level.

Examples: The first example lists the contents of a sequential file in Hex:

```
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
```

The next program segment determines program flow based upon a user's response:

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
. . .
```

INSTR Function

Format: `INSTR([i,]x$, y$)`

Action: Searches for the first occurrence of string *y$* in *x$*, and returns the position where the match occurs.

i is an offset that determines the starting position for the search. Its value may range from 1 to 255. If the value for *i* is outside this range, an `Illegal function call` occurs. When the value of *i* exceeds the number of characters in *x$*, the function returns a value of zero.

x$* and *y$ may be string variables, string expressions, or string literals.

If either *x$* is the null string or *y$* is not within *x$*, the function returns a value of zero.

When *y$* is the null string, the function returns *i* (or 1 if you omitted the offset parameter).

Example: In the following example, when the search starts at the string's beginning, the first occurrence of "`B`" is position 2. However, when an offset parameter skips the first "`B`", the function returns the position for the next occurrence (that is, position number 6):

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$); INSTR(3,X$,Y$)
RUN
 2  6
Ok
```

INT Function

Format: `INT(x)`

Action: Returns the largest integer that is less than or equal to x. See the `FIX` and `CINT` functions which also return integer results.

Examples:

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

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

KILL Command/Statement

Format: `KILL` *filename*

Purpose: Deletes the named file from disc.

Remarks: *filename* is a string expression. When *filename* is a literal, you must enclose the name in quotation marks.

filename must include the extension designator, if one exists. Although BASIC provides the `.BAS` designator as a default file type extension when you save a file, it does not supply a default designator for the `KILL` statement. For example, if you save a program with the statement:

```
SAVE "MYPROG"
```

BASIC supplies the extension `.BAS` for you. However, if you later decide to delete that program, you must supply the file's complete name as in:

```
KILL "MYPROG.BAS"
```

filename may contain question marks (`?`) or asterisks (`*`) as **wild cards**. A question mark matches any single character in the filename. An asterisk matches one or more characters, beginning from that position.

CAUTION

You should exercise extreme caution if you use wild cards with this command. See second example.

If you give the `KILL` statement for an open file, BASIC closes the file and then deletes it.

You may use the `KILL` statement for all types of disc files (program files, random data files, and sequential data files).

Example: The first example deletes `DATA 83. BAS:`

```
KILL "DATA 83.BAS"
```

The second example deletes `CHAP.1`, `CHAP.2`, and so on, but would also delete `CHAP.NEW`, `CHAP.FINAL`, and `CHAP.OUT` if these files existed:

```
KILL "CHAP.*"
```

LEFT$ Function

Format: `LEFT$(x$,i)`

Action: Returns a string comprised of the leftmost *i* characters of *x$*.

i must be in the range of 0 to 255. When *i* is greater than the number of characters in *x$*, LEFT$ returns the entire string. When *i* equals zero, the function returns the null string (a string with zero length).

Also see the MID$ and RIGHT$ functions.

Example:

```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,2)
30 PRINT B$
RUN
BA
Ok
```

LEN Function

Format: `LEN(x$)`

Action: Returns the number of characters in *x$*. LEN counts all non-printing and blank characters.

Example: In this example, because BASIC initializes all string variables to the null string, the first PRINT statement prints a value of zero:

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

LET Statement

Format: [`LET`] *variable* = *expression*

Purpose: Assigns the value of an expression to a variable.

Remarks: The reserved word `LET` is optional as the equal sign suffices when assigning an expression to a variable name.

variable is the name of a string or numeric variable that receives the value. It may be a simple variable or the element of an array.

BASIC evaluates *expression* to determine the value that it assigns to *variable*. The type for *expression* must match the *variable* type (string or numeric), or a `Type mismatch` error occurs.

BASIC interprets the leftmost equal sign in an expression to be the assignment operator. It treats subsequent equal signs as relational operators. For example, in evaluating the following expression, BASIC sets the value of `A` to true (-1) if `B` is equal to `C`.

```
A = B = C
```

Example: The first example demonstrates the use of the `LET` statement:

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

The following statements make the identical assignments but omit the word `LET`:

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

LINE INPUT Statement

Format: `LINE INPUT` [;] ["*prompt*";] *stringvar*

Purpose: Enters an entire line (up to 254 characters) to a string variable. No string delimiters are necessary.

Remarks: *prompt* is a string literal that BASIC displays upon the computer screen prior to accepting keyboard input. Including a question mark as part of the prompt requires your putting the question mark character at the end of *prompt*.

BASIC assigns all characters that occur between the end of the prompt and the end of the line to *stringvar*. (BASIC determines that a line has ended when you press the [Return] key, or it has read 254 characters.) However, if BASIC reads a linefeed/carriage return combination, both characters are echoed, but the carriage return is ignored. BASIC includes the linefeed character in *stringvar* and continues reading the input data.

When you immediately follow the reserved words `LINE INPUT` with a semicolon, pressing the [Return] key to end the input line does not echo a carriage return/line feed sequence. (That is, the cursor remains on the line where you entered your response.)

You may interrupt the entering of data to a `LINE INPUT` statement by simultaneously pressing the [CTRL] and [C] keys. BASIC returns control to the command level and issues the interpreter's `Ok` prompt. You may then use the `CONT` state to resume execution at the `LINE INPUT` statement.

Example:

```
80 LINE INPUT "CUSTOMER INFORMATION? ";C$
90 PRINT "VERIFY ENTRY: "; C$
. . .
RUN
CUSTOMER INFORMATION? BEATRICE ISOLDA 95073
VERIFY ENTRY:  BEATRICE ISOLDA 95073
```

LINE INPUT# Statement

Format: `LINE INPUT#` *filenum*, *stringvar*

Purpose: Reads an entire line (up to 254 characters) from a sequential disc data file and assigns them to the string variable. No string delimiters are required.

Remarks: *filenum* is the number you gave the file when you opened it for input.

BASIC assigns the line to *stringvar*. This parameter may be either a string variable or an array element.

The `LINE INPUT#` statement reads all characters in the sequential file up to, but not including, a carriage return character. It then skips over the carriage return (or a carriage return/ line feed sequence). The next `LINE INPUT#` statement then reads all the following characters up to the next carriage return character.

NOTE

The `LINE INPUT#` statement preserves a line feed/ carriage return sequence. For example, if a file contains the following ASCII characters:

A C_R L_F B C_R C L_F D C_R L_F E L_F C_R F C_R L_F

then the following program:

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

returns the following values to `C$`:

1st iteration: A
2nd iteration: B
3rd iteration: C L_F D
4th iteration: E L_F C_R F

You will find the LINE INPUT# statement especially useful if each line of a data file contains several fields, or if a BASIC program that was saved in ASCII mode is being read as a data file by another program.

Example:

```
10 OPEN "O", 1, "LIST"
20 LINE INPUT "BIRTH STATS? ", 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
BIRTH STATS? ELAINA MICHELLE 8 2, 20, SOQUEL
ELAINA MICHELLE 8 2, 20, SOQUEL
Ok
```

LIST and LLIST Command

Format: `LIST` [*first.line*][-[*last.line*]]

`LLIST` [*first.line*][-[*last.line*]]

Purpose: Lists all or part of the program currently in computer memory to the screen; or, if `LLIST` is used, to a line printer.

Remarks: *first.line* is the first line to be listed while *last.line* is the last line to be listed. Both must be valid line numbers within the range of 0 to 65529.

When you omit both line number parameters, the listing begins with the first line of the program and goes to the end of the program.

Specifying `first.line` prints only that line.

Specifying `first.line-` prints that line through the end of the program.

Specifying `-last.line` prints all lines from the beginning of the program through the given line.

Specifying `first.line-last.line` prints all the lines within that range.

NOTE

You may stop the listing of a program by pressing [CTRL] [C].

You may use a period (`.`) for either line number to indicate the current line. For example, you could list all the lines from the beginning of the program to the current line with this command:

```
LIST -.
```

BASIC always returns control to the command level after a `LIST` or `LLIST` command executes.

NOTE

The `LLIST` command assumes a printer line width of 132 characters.

Examples: The first example lists the program currently stored in your computer's memory:

```
LIST
```

The next statement lists only line 500:

```
LIST 500
```

The next example lists all program lines from line 50 through the end of the program:

```
LIST 50-
```

The next statement lists all program lines from the program's first line through line 50:

```
LIST -50
```

The last example lists lines 50 through 80, inclusively.

```
LIST 50-80
```

NOTE

The BASIC compiler offers no support for this command.

LOAD Command

Format: `LOAD` *filename* [`,R`]

Purpose: Loads a BASIC program file from disc into your computer's memory.

Remarks: *filename* is the string expression that you used to name the file when you saved it. When *filename* is a literal, you must enclose the name in quotation marks.

When you omit the MS-DOS file type extension from the file's name, BASIC adds the default extension `.BAS` to the filename if the name is less than nine characters.

Before it loads the named program, BASIC closes all open files and deletes all variables and program lines that currently reside in BASIC memory. However, by using the `R` option, you can run the program after it is loaded. Furthermore, all opened data files remain open. Thus, you may use the `LOAD` command with the `R` option to chain together several program (or segments of the same program). You pass information between the programs through shared data files.

Example: The first example loads and runs the program `TESTRUN`:

```
LOAD "TESTRUN",R
```

The next example loads the program `MYPROG` from the disc in drive C but does not run the program:

```
LOAD "C:MYPROG"
```

NOTE

The BASIC compiler offers no support for this command.

LOC Function

Format: `LOC(`*filenum*`)`

Action: With random-access files, `LOC` returns the record number of the last record referenced in a `GET` or `PUT` statement.

With sequential files, `LOC` returns the number of sectors (that is, 256 byte blocks) read from or written to the file since it was opened.

When you open a file for sequential input, BASIC reads the first sector of the file. Therefore, `LOC` always returns a "`1`" even before any input from the file occurs.

filenum is the number you gave the file when you opened it.

Example:

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

LOF Function

Format: `LOF(`*filenum*`)`

Action: Returns the length of the file in bytes.

filenum is the number you gave the file when you opened it.

Example: In this example, the variables `REC` and `RECSIZE` contain the record number and record length. The calculation determines whether the specified record is beyond the end-of-file.

```
90 IF REC * RECSIZE > LOF(1)
     THEN PRINT "INVALID ENTRY"
```

LOG Function

Format: `LOG(x)`

Action: Returns the natural logarithm of *x*.

x must be a positive number.

Example:

```
PRINT LOG(45/7)
 1.860752
Ok
```

LPOS Function

Format: `LPOS(x)`

Action: Returns the current position of the line printer print head within the line printer buffer. This may differ from the physical position of the print head.

x is a dummy argument.

Example:

```
100 IF LPOS(X) > 132 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Format: `LPRINT` [*list.of.expressions*]

`LPRINT USING` *stringexp*`;` *list.of.expressions*

Purpose: Prints data to a line printer.

Remarks: These statements are identical to `PRINT` and `PRINT USING`, except output goes to a line printer. For details of operation, see the `PRINT` and `PRINT USING` statements in this chapter.

`LPRINT` assumes that the printer has a line width of 132 characters.

Example:

```
LPRINT "THIS IS A TEST"
```

LSET and RSET Statements

Format: `LSET` *stringvar* = *stringexp*
`RSET` *stringvar* = *stringexp*

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

Remarks: *stringvar* is the name of a variable that you defined in a `FIELD` statement.

stringexp identifies the information that is to be placed into the field named by *stringvar*.

When *stringexp* requires fewer bytes than were allocated to *stringvar*, `LSET` left-justifies the string in the field, while `RSET` right-justifies the string. (Spaces pad the extra positions.) When a string is too long for the field, the excess characters are dropped from the right.

You must use the `MKI$`, `MKS$`, or `MKD$` function to convert numeric values to strings before you move them into the random file buffer with a `LSET` or `RSET` statement.

NOTE

You may also use `LSET` and `RSET` to left-justify or right-justify a string in a given field. For example, the following program lines right-justify the string `NOTE$` in a 20-character field:

```
110 LSET A$ = SPACE$(20)
120 RSET A$ = NOTE$
```

You will find these statements helpful when formatting output to a printer.

Example:

```
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
```

MERGE Command

Format: `MERGE` *filename*

Purpose: Incorporates statements contained in the specified file into the program that currently resides in your computer's memory.

Remarks: *filename* is the string expression that you used to name the file when you saved it. When *filename* is a literal, you must enclose the name in quotation marks.

When you omit the MS-DOS file type from the file's name, BASIC provides the default type `.BAS` for you.

You must use ASCII format when you save the file that you want to merge. (That is, you must specify the `A` option when you give the `SAVE` command.) When BASIC detects another format, it displays a `Bad file mode` error message. If this happens, BASIC cancels the `MERGE` command and the program in memory remains unchanged.

You may view the `MERGE` command as "inserting" the lines from the program on disc into the program in memory. When both programs have identical line numbers, the lines from the disc file replace the corresponding lines in memory.

Example: This example shows how the merge command replaces or adds lines to the program currently in memory based upon each program's line numbers.

(Merge File = FILE2)

```
15 REM THIS FILE CHANGES THE LOOP CONTENTS
30       COUNT = COUNT + I
40       PRINT COUNT

LOAD "FILE1" [Return]
LIST [Return]
10 REM THIS FILE IS THE RESIDENT FILE
20 FOR I = 1 TO 10
30       PRINT "HELLO";
50 NEXT I
60 PRINT "DONE"
Ok
MERGE "FILE2" [Return]
Ok
LIST [Return]
10 REM THIS FILE IS THE RESIDENT FILE
15 REM THIS FILE CHANGES THE LOOP CONTENTS
20 FOR I = 1 TO 10
30       COUNT = COUNT + I
40       PRINT COUNT
50 NEXT I
60 PRINT "DONE"
Ok
```

NOTE

The BASIC compiler offers no support for this command.

MID$ Function

Format: `MID$(x$,i[,j])`

Action: Returns a string of length *j* characters that begins with the *ith* character in string *x$*.

x$ is any string expression.

i is an integer expression that may range between 1 to 255. *j* is an integer expression that may range between 0 and 255. Numbers outside these ranges produce an `Illegal function call.`

When you omit the length parameter *j*, or if fewer than *j* characters exist to the right of the *ith* character, `MID$` returns all the characters beginning with the *ith* character.

When the starting point *i* exceeds the length of *x$*, `MID$` returns the null string.

Also see the `LEFT$` and `RIGHT$` functions.

Example:

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

MID$ Statement

Format: `MID$(x$,i[,j]) = y$`

Purpose: Replaces a portion of one string with another string.

Remarks: *x$* is a string variable or an array element. BASIC replaces the designated characters of this string.

i is an integer expression that may range from 1 to 255. It marks the starting position in *x$* where replacement begins.

j is an integer expression that may range from 0 to 255. It gives the number of characters from *y$* that BASIC uses in the replacement. When you omit this parameter, BASIC uses the entire *y$* string.

NOTE

The length of *x$* is fixed. Therefore, if *x$* is five characters long and *y$* is ten characters long, BASIC only replaces *x$* with the first five characters of *y$*.

Example:

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

MKI$, MKS$, MKD$ Functions

Format: `MKI$ (`*integer.expression*`)`
`MKS$ (`*single-precision.expression*`)`
`MKD$ (`*double-precision.expression*`)`

Action: Converts numeric values to string values.

Random-access disc files store numeric values as strings. Therefore, when you place values in a random disc file by using the `LSET` or `RSET` statement, you must first convert the numbers to strings.

`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 a 8-byte string.

See also `CVI`, `CVS`, and `CVD` for the complementary operations.

Example:

```
100 AMT = (K+T)
110 FIELD #1, 8 AS D$, 20 AS N$
120 LSET D$ = MKS$(AMT)
130 LSET N$ = A$
140 PUT #1
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive versions of these functions.

NAME Statement

Format: `NAME` *oldname* `AS` *newname*

Purpose: Changes the name of a file to the newly given name.

Remarks: *oldname* is a string expression for the name you gave the file when you opened it or saved it.

newname is also a string expression that conforms to the rules for a valid filename. If the file is a `.BAS` file, you must include the file type `.BAS` in the file's name. BASIC does not supply `.BAS` as a default type for you.

When either *oldname* or *newname* is a literal, you must enclose the string in quotation marks.

A file must exist with *oldname*. Similarly, no file can exist with *newname*. When BASIC fails to find *oldname*, it gives a `File not found` error. Likewise, if BASIC finds that a file already exists with *newname*, it displays the message `File already exists`.

oldname must be closed before the renaming operation.

If *oldname* and *newname* contain a drive designator, the drive must be the same. Attempting to rename a file on a different disc produces a `Rename across disks` error.

A free file handle must exist for performing the open check. Otherwise, a `Too many files` error occurs.

Example: The following statement changes the name of the file `ACCTS` to `LEDGER` on drive C. After the `NAME` statement executes, the file still resides on the same area of disc space on the same disc, but with the new name.

```
NAME "C:ACCTS" AS "C:LEDGER"
```

NEW Command

Format: `NEW`

Purpose: Deletes the program that currently resides in computer memory and clears all variables.

Remarks: You use the `NEW` command in Direct Mode to clear extraneous information from your computer's memory before you enter a new program.

You must enter the `NEW` command at the command level. Control remains at the command level after this statement executes.

Example:

```
Ok
NEW
Ok
```

NOTE

The BASIC compiler offers no support for this command.

NULL Statement

Format: `NULL` *integer.expression*

Purpose: Sets the number of nulls that BASIC prints at the end of each line. This number applies to both the display and a printer.

Remarks: *integer.expression* is the number of null characters (00 Hex) that BASIC appends at the end of each line. The default setting is zero.

The ASCII characters between 00 Hex and 20 Hex are called Control Characters. (For example, this range includes the backspace character, carriage return character, and line feed character.) As some devices take longer to process certain control characters, they require an extra amount of time before they receive the next significant character.

When using Hewlett-Packard peripherals, you may omit using the `NULL` statement.

Example: `NULL 2`

OCT$ Function

Format: `OCT$(x)`

Action: Returns a string that represents the octal value of the decimal argument. BASIC rounds *x* to an integer before it evaluates `OCT$(X)`.

See the `HEX$` function for hexadecimal conversion.

Example:

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

ON ERROR GOTO Statement

Format: `ON ERROR GOTO` *line*

Purpose: Enables error trapping and specifies the first line of the error-handling subroutine.

Remarks: *line* is the line number of the first line of an error-handling routine. If the line number does not exist, an `Undefined line number` error occurs.

Once you have enabled error trapping, BASIC sends program control to the specified line number whenever it detects an error. (This also includes Direct Mode errors, such as syntax errors.)

You use the `RESUME` statement to leave an error-handling routine.

You may disable error trapping by executing an `ON ERROR GOTO 0` statement. Any subsequent errors print an error message and halt execution. Within an error-trapping subroutine, the `ON ERROR GOTO 0` statement halts BASIC and prints the error message for the error that triggered the trap. We recommend that all error-trapping subroutines execute an `ON ERROR GOTO 0` statement if an error is encountered for which no recovery action exists.

NOTE

If an error occurs during execution of an error-handling subroutine, BASIC prints an error message and halts execution. Further error trapping does not occur within a error-handling subroutine.

Example: The following program segments illustrate the effects of the ON ERROR and RESUME statements:

```
 5 REM Example without RESUME
10 ON ERROR GOTO 40
20 Y = 9 : Z = 0
30 L = 30 : X = Y/Z 'Division by zero
40 PRINT "ERROR ENCOUNTERED IN LINE "; L
50 END
RUN
ERROR ENCOUNTERED IN LINE 30
Ok
 8 REM With RESUME, execution continues
 9 REM on line where the error occurred
10 ON ERROR GOTO 60
20 Y = 9 : Z = 0
30 L = 30 : X = Y/Z
40 PRINT "CONTINUE PROGRAM"
50 GOTO 90
60 PRINT "ERROR ENCOUNTERED IN LINE "; L
70 Z = 5
80 RESUME
90 PRINT "END"
100 END
RUN
ERROR ENCOUNTERED IN LINE 30
CONTINUE PROGRAM
END
Ok
```

While in Direct Mode, all errors default to the ON ERROR statement:

```
30 PRINT "THIS SYNTAX IS NO GOOD!!"
ON ERROR GOTO 30
Ok
PRING "ERROR"
THIS SYNTAX IS NO GOOD!!
No RESUME in 30
Ok
```

NOTE

If you plan to compile a program that uses the ON ERROR GOTOstatement, please refer to the BASIC compiler manual. Also, set the compiler switches properly so your event trapping routine works correctly.

ON. . .GOSUB Statement

Format: `ON` *result* `GOSUB` *line* [`,`*line*] . . .

Purpose: Branches to a subroutine or subroutines depending upon which value is returned from the governing expression.

Remarks: *result* is a numeric expression which must return a value between 0 and 255. (BASIC rounds the expression to an integer value when necessary.) Any value outside this range causes an `Illegal function call` error.

line is the beginning line number for a subroutine.

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

When the value of *result* is zero or greater than the number of items in the list, BASIC continues with the next executable statement.

Example:

```
20 INPUT "ENTER TRIG FUNCTION"; A$
30 IF A$ = "SIN" THEN F = 1 : GOTO 70
40 IF A$ = "COS" THEN F = 2 : GOTO 70
50 IF A$ = "TAN" THEN F = 3 : GOTO 70
60 PRINT "ILLEGAL ENTRY TRY AGAIN" : GOTO 20
70 FOR K = 0 TO 360 STEP 10
80     PRINT K;
90     A = K/180*3.14159
100    ON F GOSUB 1000, 2000, 3000
110 NEXT K
120 STOP
999 REM SUBROUTINE SECTION
1000 PRINT SIN(A) : RETURN
2000 PRINT COS(A) : RETURN
3000 PRINT TAN(A) : RETURN
```

ON. . .GOTO Statement

Format: `ON` *result* `GOTO` *line* [,*line*] . . .

Purpose: Branches to one of several specified line numbers, depending upon which value BASIC returns when it evaluates the controlling expression.

Remarks: *result* is a numeric expression which must return a value between 0 and 255. (BASIC rounds the expression to an integer value when necessary.) Any value outside this range causes an `Illegal function call` error.

line is the line number where you want program control to go.

The value of *result* determines to which line number program control branches. For example, if the returned value were 3, program control branches to the third line number in the list.

When the value of *result* is zero or greater than the number of items in the list, BASIC continues with the next executable statement.

Example:

```
10 REM Simple selection program
20 INPUT "ENTER SELECTION FROM MENU"; K
30 ON K GOTO 50, 70, 90
40 PRINT "INVALID SELECTION" : GOTO 20
50 PRINT "YOU CHOSE SELECTION NUMBER 1"
60 GOTO 20
70 PRINT "YOU CHOSE SELECTION NUMBER 2"
80 GOTO 20
90 PRINT "YOU CHOSE 3 TO END THIS PROGRAM"
100 END
RUN
ENTER SELECTION FROM MENU? 0 [Return]
INVALID SELECTION
ENTER SELECTION FROM MENU? 2 [Return]
YOU CHOSE SELECTION NUMBER 2
ENTER SELECTION FROM MENU? 3 [Return]
YOU CHOSE 3 TO END THIS PROGRAM
Ok
```

OPEN Statement

Format 1: `OPEN` *filename* [`FOR` *mode*] `AS` [`#`] *filenum* [`LEN=`*recl*]

Format 2: `OPEN` *mode2*`,` [`#`] *filenum*`,` *filename* [`,` *recl*]

Purpose: Grants access to a file for reading or writing.

Remarks: In Format 1, *mode* can be:

`INPUT`	for sequential input mode
`OUTPUT`	for sequential output mode
`APPEND`	for sequential output mode. Additionally, BASIC positions the file to the end of the data when you open the file.

When you omit the *mode* parameter, the program assumes random access.

NOTE

Even though *mode* is a string constant, you must not enclose the string in quotation marks.

In Format 2, *mode2* can be:

`I`	for sequential input mode
`O`	for sequential output mode
`R`	for random input or output

Disc files allow all modes.

filename is a string expression that names the file. It may include a file type (`.xxx`) and a drive specifier if the file is not on the current disc. When *filename* is a literal, you must enclose the string in quotation marks (`"`).

filenum is an integer expression that gives that file's identifying number. Its value may range from 1 to the maximum number of files allowed. The normal maximum setting is 3, but you may change this value with the `/F:` switch on the BASIC command line.

Once you assign a number to the file, BASIC associates this number to that file for as long as the file remains open. You use *filenum* when using other disc I/O statements with the file.

recl is an integer expression that sets the record length. You can define *recl* for random-access files. The default is 128 bytes. The value you use for *recl* must not exceed the value you set on the BASIC command line for the `/S:` switch when you initialized BASIC.

NOTE

You may also set the maximum record length by using the `/S` option when initializing BASIC with the MS-DOS `BASIC` command. However, you cannot use this option with sequential files.

A program must execute an `OPEN` statement before you can use any of the following commands:

`PRINT#`, `PRINT# USING`, `INPUT#`, `LINE INPUT#`

`WRITE#`, `INPUT#`, and `GET` & `PUT`

You must open a disc file before you can perform any read or write operation on that file.

The `OPEN` statement allocates an I/O buffer to the file and determines the buffer's mode of access.

You may open a file for sequential input or random access on more than one file number at a time. You may only open a file for output, however, on one file number at a time.

Examples: This program segment accepts input to an inventory file:

```
10 OPEN "I", 2, "INVEN"
20 INPUT #2, PART$, DESC$
30 PRINT PART$; DESC$
40 GOTO 20
```

The next example opens the file `MAIL.DAT` so data is added to the end of the file:

```
10 OPEN "MAIL.DAT" FOR APPEND AS 1
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences in the interpretive and compiled versions of this statement.

OPTION BASE Statement

Format: `OPTION BASE` *n*

Purpose: Sets the minimum value for array subscripts.

Remarks: *n* may be either 1 or 0.

BASIC normally numbers arrays from a base of zero. When you want an array index to begin at 1, you must use the `OPTION BASE` statement.

If you decide to use the `OPTION BASE` statement, you must include it within your program **before** you define or use any arrays.

Example: This example sets up a string array with ten elements (1..10) and a numeric array with 20 elements (1..20):

```
10 OPTION BASE 1
20 DIM LNAME$, ID(20)
```

OUT Statement

Format: `OUT` *i*, *j*

Purpose: Sends a byte to the specified output port.

Remarks: *i* is an integer expression that ranges between 0 and 65535. It is a microprocessor port number.

NOTE

The output port is a microprocessor port. It does not refer to your computer's datacomm (or peripheral) ports.

---

j is an integer expression that ranges between 0 to 255. It is the byte of data that you want to send. For example, a zero sets all eight bits to zeroes while 255 sets all eight bits to ones.

`OUT` is the complementary command to the `INP` function.

Example:

```
100 OUT 12345, 255
```

PEEK Function

Format: `PEEK(i)`

Action: Returns the byte read from memory location *i*.

The result is a decimal integer that ranges between 0 (eight zeros) to 255 (eight ones).

i must be within the range of -32768 to 65535. (It is an offset from the current segment, which you set with the `DEF SEG` statement.) When the function returns a negative value, you should add 65536 to that value to obtain the actual address.

`PEEK` is the complementary function to the `POKE` statement.

Example:

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

POKE Statement

Format: POKE *address*, *data*

Purpose: Writes a byte of information into a memory location.

Remarks: *address* is an integer expression for the address of the memory location to be poked. (It is an offset from the current segment, which you set with the DEF SEG statement.) The value must be within the range of 0 to 65535.

data is an integer expression for the data to be poked. It must be within the range of 0 (which would set all eight bits to zeroes) to 255 (which would set all eight bits to ones).

PEEK is the complementary function to POKE. PEEK's argument is an address from which a byte of information is read.

You can use PEEK and POKE for efficiently storing data, loading assembly-language subroutines, and passing arguments and results to and from assembly-language subroutines.

CAUTION

BASIC does not check the address. Therefore, use this statement with extreme care so you do not inadvertently overwrite meaningful data.

Example: This example places hex value FF (decimal 255, or a byte with 1's in all eight positions) into the Data Segment relative memory location at hex 5A00:

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

POS Function

Format: `POS(0)`

Action: Returns the cursor's current column position. The leftmost column is position number 1. The rightmost column is position number 80.

`0` is a dummy argument.

See also the `LPOS` function and the `WIDTH` statement.

Example:

```
IF POS(0) > 60 THEN PRINT CHR$(13)
```

PRINT Statement

Format: PRINT [*list.of.expressions*]

Purpose: Copies data to the computer screen.

Remarks: *list.of.expressions* is a list of numeric and/or string expressions. You must separate multiple items with commas, blanks, or semicolons and enclose any string constants with quotation marks.

Including *list.of.expressions* prints the values of those expressions on the screen.

Omitting *list.of.expressions* prints a blank line.

Print Positions: The punctuation symbols that separate the listed items determine the position where BASIC prints each item.

BASIC divides the line into **print zones** of 14 spaces each. Within *list.of.expressions*, a comma prints the next value at the beginning of the next zone. A semicolon prints the next value immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

When a comma or semicolon ends the list of expressions, the next PRINT statement continues printing on the same line, spacing accordingly. If the list ends with no comma or semicolon, BASIC ends the line by printing a carriage return character. (That is, it advances the cursor to the next line.)

When the printed line exceeds the width of the screen, BASIC wraps the line to the next physical line and continues printing.

For numbers, BASIC reserves the first character position for a numeric sign. It precedes positive numbers with a space. It precedes negative numbers with a minus sign. BASIC always prints a space as a separator after any number.

You may enter a question mark (?) as an abbreviation for the word PRINT in a PRINT statement. When BASIC lists the program, it automatically replaces the question mark with the reserved word PRINT.

To send output to a line printer, use the `LPRINT` and `LPRINT USING` statements.

NOTE

When single-precision numbers can be represented with 7 or fewer digits in unscaled format no less accurately than they can be represented in scaled format, BASIC prints the numbers using unscaled format (either integer or fixed point). For example, BASIC prints `1E-7` as `.0000001` whereas it prints `1E-8` as `1E-08`.

When double-precision numbers can be represented with 16 or fewer digits in unscaled format no less accurately than they can be represented in scaled format, BASIC prints the numbers using the unscaled format. For example, BASIC prints `1D-16` as `.0000000000000001` whereas it prints `1D-17` as `1D-17`.

Examples: The commas in the following PRINT statement prints each succesive value at the next print zone:

```
10 X = 5
20 PRINT X+5, X-5, X*5, X/5
30 END
RUN
 10       0        25       1
Ok
```

In the following program segment, the semicolon at the end of line 20 prints the information from lines 20 and 30 on the same line. Line 40 prints a blank line before the next prompt:

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

? 21 [Return]
 21 SQUARED IS 441 AND 21 CUBED IS 9261

? [CTRL] [C]
```

In the following example, the semicolons in the PRINT statement print each value immediately after the preceding value. Remember, positive numbers are preceded by a space, and all numbers are followed by a space. Line 40 uses the question mark as an abbreviation for PRINT:

```
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
```

PRINT USING Statement

Format: PRINT USING *stringexp*; *list.of.expressions*

Purpose: Uses a specified format to print strings or numbers.

Remarks and Examples:

list.of.expressions contains the string or numeric expressions that you want to print. You must separate the items in the list with commas or semicolons.

stringexp is either a string constant or a string variable that is comprised of special formatting characters. These formatting characters (see below) determine the field and format of the printed strings or numbers.

When entering program lines, you may use a question mark (?) as an abbreviation for the reserved word PRINT. BASIC automatically replaces this symbol with PRINT when you list the program.

String Fields: When you use the PRINT USING statement to print strings, you may select one of three characters to format the string field:

! An exclamation point limits printing to the first character in the string.

\n spaces\ Two back slash characters separated by n spaces prints that number of characters (that is, $n+2$). For example, typing just the backslashes prints two characters; typing one space between the backslashes prints three characters; and so on. When the field is longer than the string, BASIC left-justifies the string within the field and pads the remainder of the field with spaces. Consider this example:

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

&   An ampersand specifies a variable length string field. Using this formatting character echoes the string exactly as you entered it.

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

Numeric Fields:

When printing numbers with the `PRINT USING` statement, you may use the following special characters to format the numeric field.

The number sign signifies a digit position. BASIC fills in all requested digit positions. When a number has fewer digits than the positions specified, BASIC right-justifies the number in the field (that is, leading unused positions are replaced with spaces).

You may insert a decimal point at any position within the field. When the format string specifies that a digit should appear before the decimal point, BASIC always prints a digit (0 if necessary). BASIC also rounds numbers as required to fit the format.

Consider these examples:

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

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

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

In the last example, the three spaces at the end of the format string provide spacing between the printed values.

+ A plus sign at the beginning or end of the format string prints the sign of the number (plus or minus) before or after the number, depending upon the placement of the plus sign in the format string.

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

- A minus sign at the end of the format field prints a trailing minus sign after negative numbers.

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

** A double asterisk at the beginning of the format string replaces leading spaces with asterisks. The double asterisk also reserves two more digit positions.

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

$$ A double dollar sign prints a dollar sign to the immediate left of the formatted number. The double dollar symbol reserves two more digit positions, one of which is the dollar sign. You cannot use the exponential format in conjunction with $$. Furthermore, you can print negative dollar amounts only if the minus sign trails to the right.

```
PRINT USING "$$###.##-"; 456.78, -45.54
 $456.78   $45.54-
```

**$ Placing **$ at the beginning of a format string combines the effects of the two previous symbols. BASIC replaces leading spaces with asterisks and prints a dollar sign before the number. Additionally, **$ reserves three digit positions, one of which is used for the dollar sign.

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

,

A comma that appears to the left of the decimal point in a formatting string prints a comma as a thousands separator. When the comma appears at the end of the formatting string, the comma is printed following the number. The comma represents another digit position. It has no effect when used with the exponential format (^^^^).

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

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

^^^^

You may place four carets (or circumflexes) after the digit position characters to specify exponential format. The four carets reserve space to print `E+xx` (or `D+xx`). Any decimal point position may be specified. BASIC left-justifies the significant digits and adjusts the exponent accordingly. Unless you include either a plus formatting character or a trailing plus or minus formatting character, BASIC reserves one space to the left of the decimal point to print a space (for positive numbers) or a minus sign (for negative numbers).

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

```
PRINT USING ".###^^^^-"; -88888
.889E+05-
```

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

_

An underscore character in the format string prints the next character as a literal character.

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

You may include the underscore character within the formatting string by preceding it with an underscore. The next example contains a string constant within the format string.

```
PRINT USING "EXAMPLE __#"; 1
EXAMPLE _1
```

BASIC prints a percent sign (`%`) before a number when the printed value exceeds the specified numeric field. When rounding causes the number to exceed the field length, BASIC prints the percent sign before the rounded number.

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

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

If the number of digits exceeds 24, an `Illegal function call` results.

PRINT# and PRINT# USING Statements

Format: `PRINT#` *filenum*`,` [`USING` *stringexp*`;`] *list.of.expressions*

Purpose: Writes data to a sequential disc file.

Remarks: *filenum* is the number you gave the file when you opened it for output.

stringexp consists of the formatting characters as described for the `PRINT USING` statement.

The expressions in *list.of.expressions* are the numeric and/or string values that you want to write to the file.

`PRINT#` does not compress data on the disc. With this statement, BASIC writes an image of the data to disc, just as it would display the information on your computer screen. For this reason, you must carefully delimit the data on the disc so that future input statements can correctly read the data.

In *list.of.expressions,* you should separate all numeric expressions with semicolons (`;`). For example,

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

If you use commas to separate the expressions, BASIC copies the extra blanks between the print fields to the disc file.

You must separate string expressions in the list with semicolons. 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$;B$
```

writes the following data to the disc:

```
CAMERA93604-1
```

Since the `PRINT#` statement omitted explicit delimiters, you would be unable to use an `INPUT#` statement to read both strings back in. To correct this problem, insert explicit delimiters into the `PRINT#` statement as follows:

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

This statement writes the following image to disc:

```
CAMERA,93604-1
```

In this form, you may use the `INPUT#` statement to read both values.

When the strings themselves contain commas, semicolons, significant leading spaces, carriage return, or line feed characters, you must surround the string with explicit quotation marks, that is `CHR$(34)`.

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

The statement:

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

writes the following image to disc:

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

Therefore, the following `INPUT#` statement:

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

assigns `"CAMERA"` to `A$` and `"AUTOMATIC 93604-1"` to `B$`.

To separate these strings properly on the disc, include double quotes within the string by 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"
```

Therefore, the statement:

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

assigns `"CAMERA, AUTOMATIC"` to `A$` and `"     93604-1"` to `B$`.

You may also use the `PRINT#` statement with the `USING` option to format the data printed to the disc file. For example,

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

See `WRITE#` for more examples.

PUT Statement

Format: `PUT [#]` *filenum* `[,` *recnum*`]`

Purpose: Writes a record from the random file buffer to a random-access disc file.

Remarks: *filenum* is the number you gave the file when you opened it.

recnum identifies the record to be written. It may range from 1 to 32767.

When you omit *recnum*, BASIC uses the next available record number (after the last `PUT`).

NOTE

You may use `PRINT#`, `PRINT# USING`, and `WRITE#` to put characters in the random file buffer before a `PUT` statement executes. When you use the `WRITE#` statement, BASIC pads the buffer with spaces up to the carriage return character. Attempting to read or write beyond the end of the buffer causes a `Field overflow` error.

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
```

RANDOMIZE Statement

Format: RANDOMIZE [*expression*]

Purpose: Reseeds the random-number generator.

Remarks: When you omit *expression*, BASIC suspends program execution and asks for a value by printing:

```
Random number seed (-32768 to 32767)?
```

After you enter a value, BASIC executes the RANDOMIZE statement.

If you fail to reseed the random-number generator the RND function returns the same sequence of "random" numbers each time you run the program. To change the seed each time the program runs, place a RANDOMIZE statement at the beginning of the program and change its argument before each run.

Example:

```
10 RANDOMIZE
20 FOR I = 1 TO 5
30     PRINT RND;
40 NEXT I
50 END
RUN
Random number seed (-32768 to 32767)?
```

(you type 3 [Return])

```
.2226007 .5941419 .2414202 .2013798
   5.361748E-02
Ok
```

```
RUN
Random number seed (-32768 to 32767)?
```

(you type 4 [Return])

```
 .628988  .765605  .5551516  .775797 .7834911
Ok
RUN
Random number seed (-32768 to 32767)?
```

(you type 3 [Return] which produces the first sequence)

```
.2226007  .5941419  .2414202  .2013798
   5.361748E-02
Ok
```

READ Statement

Format: `READ` *variable* [`,` *variable*] . . .

Purpose: Reads values from `DATA` statements and assigns these values to the named variables.

Remarks: *variable* is a numeric or string variable that receives the value read from a `DATA` statement. It may be a simple variable or an array element.

You always use `READ` statements in conjunction with `DATA` statements. `READ` statements assign `DATA` items to variables on a one-to-one basis. The `READ`-statement variables may be numeric or string. The values in the `DATA` statement must agree, however, with the specified variable types. If they differ, a `Syntax error` occurs.

A single `READ` statement may access one or multiple `DATA` statements, or several `READ` statements may access the same `DATA` statement. If the number of variables exceeds the number of elements in the `DATA` statement(s), BASIC prints an `Out of DATA` error message. If the number of variables is less than the number of elements in the `DATA` statement, subsequent `READ` statements begin reading data at the point where the last `READ` operation finished. When no subsequent `READ` statements occur, BASIC ignores the extra data.

You may reread `DATA` statements by using the `RESTORE` statement. (See the `RESTORE` statement for more information.)

Examples: This example reads the values from the `DATA` statements into the array `A`. After the `FOR` loop, the value of `A(1)` is 3.08, `A(2)` is 5.19, and so on:

```
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 following program segment reads both string and numeric data:

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

Note that you may omit placing quotation marks around the string `COLORADO` since it contains no commas, semicolons, or significant spaces. However, you must place quotation marks around `DENVER,` because of the comma.

This program reads string and numeric data from two consecutive `DATA` statements until all variables have been assigned a value. The excess data is ignored:

```
10 FOR K = 1 TO 5
20     READ A$ : PRINT A$;
30 NEXT K
40 DATA "TONI,", "NICO,"
50 DATA "BOB,", BERNADETTE, 52, 50, PRINGLE
60 END
RUN
TONI,NICO,BOB,BERNADETTE52
```

REM Statement

Format: `REM` *remark*

Purpose: Inserts explanatory remarks into a program without affecting program execution.

Remarks: *remark* may be any sequence of characters.

BASIC prints `REM` statements exactly as you entered them when you list the program. `REM` statements are never executed.

You may branch to a `REM` statement from a `GOTO` or `GOSUB` statement. In this case, execution continues with the first executable statement after the `REM` statement.

You may append remarks at the end of a program line by preceding the remark with a single quotation mark or apostrophe (`'`) instead of `:REM`. However, you must avoid using this method at the end of a `DATA` statement. In this event, BASIC would interpret the remark as part of the data.

NOTE

Never append programming statements to a `REM` line since BASIC will interpret the statements as part of the remark. For example, the following statements do not print a blank line:

```
500 REM Begin New Section : PRINT
```

Rather, make the `REM` statement the last statement in the line:

```
500 PRINT : REM Begin New Section
```

Examples: The first example uses the REM statement as a header for the FOR...NEXT loop:

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

The next example shows the use of the apostrophe (') for REM:

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

The last example attaches the comment to the end of the first statement of the FOR loop:

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

RENUM Command

Format: `RENUM` [[*newnumber*] [,[*oldnumber*] [,*increment*]]]

Purpose: Renumbers the lines within a program.

Remarks: *newnumber* is the first line number in the new sequence. When you omit this parameter, BASIC sets the value to 10.

oldnumber is the line in the current program where renumbering begins. When you omit this parameter, BASIC begins with the first line in the program.

increment is the amount by which the numbering increases at each step. The default value is 10.

`RENUM` also changes all references to line numbers in `GOTO`, `GOSUB`, `THEN`, `ON...GOTO`, `ON...GOSUB`, and `ERL` statements to reflect the new line numbers. When BASIC detects a nonexistent line number after one of these statements, the error message `Undefined line xxxxx in yyyyy` appears. `RENUM` leaves the incorrect line number reference `xxxxx` as it was. However, the reference to line number `yyyyy` may have changed.

CAUTION

Numeric constants following an `ERL` variable in a given expression may be treated as line references and thus modified by a `RENUM` statement. To avoid this problem, you should use statements similar to these:

```
L = ERL : PRINT L/10
```

rather than this statement:

```
PRINT ERL/10
```

You cannot use RENUM to change the order of program lines. For example, if a program contains three lines numbered 10, 20, and 30, attempting to change line 30 to line 15 to produce the new sequence 10, 15, 20 with the statement

```
RENUM 15,30
```

is illegal.

You cannot create line numbers greater than 65529. Attempting to do so causes an `Illegal function call`.

Examples: The first example renumbers the entire program. The first line number is 10 and following line numbers are incremented by 10:

```
RENUM
```

The next example also renumbers the entire program. However, the first line number is 300, and subsequent lines are incremented by 50:

```
RENUM 300,,50
```

The last example renumbers the lines beginning from 900 so they start at 1000 and increase by 20 at each step:

```
RENUM 1000,900,20
```

NOTE

The BASIC compiler offers no support for this command.

RESET Command/Statement

Format: `RESET`

Purpose: Closes all disc files and writes the directory information to every disc with open files.

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

All files must be closed before you remove a disc from its drive.

BASIC always returns to the command level after executing a `RESET` command.

RESTORE Statement

Format: RESTORE [*line#*]

Purpose: Permits a program to reread DATA statements

Remarks: After a program executes a RESTORE statement, the next READ statement accesses the first item in the program's first DATA statement. If you specify *line#*, however, the next READ statement accesses the first item in the given DATA statement.

Examples: This program segment produces an Out of DATA error:

```
10 READ A,B,C
20 READ D,E,F
30 DATA 57,68,79
40 PRINT A;B;C;D;E;F
50 END
RUN
Out of DATA in 20
Ok
```

Adding a RESTORE statement between lines 10 and 20 assigns a value to all six variables:

```
10 READ A,B,C
15 RESTORE
20 READ D,E,F
30 DATA 57,68,79
40 PRINT A;B;C;D;E;F
50 END
RUN
 57  68  79  57  68  79
Ok
```

RESUME Statement

Format:

```
RESUME
RESUME 0
RESUME NEXT
RESUME line#
```

Purpose: Continues program execution after BASIC has performed an error recovery procedure.

Remarks: You select between the various formats depending upon where you want execution to resume.

`RESUME` or `RESUME 0`	Execution resumes at the statement that caused the error.
`RESUME NEXT`	Execution resumes at the statement that immediately follows the one that caused the error.
`RESUME` *line#*	Execution resumes at *line#*.

A `RESUME` statement that is not in an error-handling routine causes a `RESUME without error` error message.

BASIC always returns to the command level after executing a `RESUME` statement.

Example:

```
80 ON ERROR GOTO 900
. . .
900 IF (ERR=230) AND (ERL=90)
      THEN PRINT "PRESS RETURN TO CONTINUE"
910 RESUME 80
. . .
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between implementations.

RETURN Statement

Format: `RETURN`

Purpose: Returns program control to the line immediately following the most recently executed `GOSUB` or `ON...GOSUB` statement.

Remarks: See the `GOSUB` and `ON...GOSUB` statements in this chapter for an example on the `RETURN` statement.

NOTE

If you plan to compile your program, check the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

RIGHT$ Function

Format: `RIGHT$(x$,i)`

Action: Returns the rightmost *i* characters of string *x$*. When *i* is equivalent to the number of characters in *x$*, `RIGHT$` returns *x$*. When *i* is zero, the function returns the null string (a string of zero length).

Also see the `MID$` and `LEFT$` functions.

Example:

```
10 A$ = "BASIC"
20 PRINT RIGHT$(A$,3)
RUN
SIC
Ok
```

RND Function

Format: `RND[(x)]`

Action: Returns a random number between 0 and 1. `RND` generates the same sequence of "random" numbers each time a program runs unless you use the `RANDOMIZE` statement to reseed the random-number generator. However, a negative value for *x* always restarts the same sequence for any given *x*.

Setting *x* to 0 repeats the last number that was generated.

Omitting *x* or specifying a positive *x* generates the next random number in the sequence.

Example:

```
10 FOR I = 1 TO 5
20      PRINT INT (RND * 100);
30 NEXT
RUN
 12     65     86     72     79
Ok
```

RUN Command/Statement

Format 1: `RUN` [*line#*]

Purpose: Executes the program currently stored in your computer's memory.

Remarks: When you include *line#*, execution begins on that line. Otherwise, execution begins with the lowest line number. BASIC always returns control to the command level when program execution finishes.

Format 2: `RUN` *filename* [`,R`]

Purpose: Loads a file from disc into your computer's memory and then executes it.

Remarks: *filename* is the name you gave the file when you saved it. (You may omit the MS-DOS file type `.BAS`, as BASIC supplies it for you.)

`RUN` closes all open files and deletes the current contents of computer memory before loading the named program. However, when you use the `R` option, all data files remain open.

For further information on files, see Chapter 4.

Example: The first example executes the program currently in memory:

```
RUN
```

The next example loads the program `NEWFIL` from disc then runs it while keeping data files open:

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

The last example uses `RUN` as a statement to re-execute the current program from its beginning:

```
9999 RUN        'Re-run program
```

NOTE

Differences exist between the interpretive and compiled version of the `RUN` command. See the BASIC compiler manual if you plan to compile your program.

SAVE Command

Format: `SAVE` *filename* [{`,A` | `,P`}]

Purpose: Stores a program file from your computer's memory to disc.

Remarks: *filename* is a quoted string that "names" the file for future references.

When the filename is less than nine characters and if you omit a file extension, BASIC supplies the default file type `.BAS` for you.

BASIC normally writes the file to the currently active disc. Saving a file to another disc requires your including a drive specifier as part of *filename*.

When a file already exists on the disc with *filename*, BASIC overwrites it. No warning is given.

The `A` option saves the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary form. ASCII format uses more disc space, but some disc accesses require that files be in ASCII format. For instance, the `MERGE` command requires ASCII formatted files. Also, any programs that you save in ASCII format may be read as data files.

The `P` option protects the file by saving it in an encoded binary format. When the protected file is later loaded or runned, any attempt to list or edit it fails. No command exists to "unprotect" such a file.

Examples: The first example saves the program `MYPROG` in ASCII format:

```
SAVE "MYPROG", A
```

The next command saves the program `STATS` as a protected file that cannot be altered:

```
SAVE "STATS", P
```

The last example saves the program `BDGT` to the disc on drive C:

```
SAVE "C:BDGT"
```

SGN Function

Format: `SGN(x)`

Action: If *x* is positive, `SGN` returns 1.
If *x* is equal to zero, `SGN` returns 0.
If *x* is negative, `SGN` returns -1.

Example:

```
10 INPUT X
20 ON SGN(X) + 2 GOTO 30, 40, 50
30 PRINT "X<0" : GOTO 60
40 PRINT "X=0" : GOTO 60
50 PRINT "X>0"
60 END
```

SIN Function

Format: `SIN(x)`

Action: Returns the sine of *x*, where *x* is given in radians.

BASIC evaluates `SIN(X)` with single-precision arithmetic.

NOTE

To convert degrees to radians, multiply the angle by PI/180, where PI = 3.141593.

Example:

```
PRINT SIN (1.50)
 .9974951
Ok
```

SPACE$ Function

Format: `SPACE$(x)`

Action: Returns a string of *x* spaces, where *x* may range between 0 and 255.

When necessary, BASIC rounds *x* to an integer.

Also see the `SPC` function.

Example:

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

SPC Function

Format: `SPC(j)`

Action: Prints *j* blanks. You may only use the `SPC` statement with the `PRINT` or `LPRINT` statements.

j is the number of spaces to be printed. When *j* is negative, `SPC` prints the null string. When *j* is greater than 255, `SPC` prints the number of blanks equal to `J MOD 255`.

`SPC` rounds floating point numbers to an integer value to determine the number of blanks to print.

Also see the `SPACE$` function.

Example: In the following `PRINT` statement, BASIC assumes that a semicolon follows `SPC(15)`:

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

SQR Function

Format: `SQR(x)`

Action: Returns the square root of *x*. *x* must be a positive number or zero.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20     PRINT X, SQR(X)
30 NEXT
40 END
RUN
 10                 3.162278
 15                 3.872984
 20                 4.472136
 25                 5
Ok
```

STOP Statement

Format: `STOP`

Purpose: Ends program execution and returns control to the command level.

Remarks: You normally use this statement when debugging a program. However, you may use `STOP` statements anywhere within a program to stop execution. Upon encountering a `STOP` statement, BASIC prints the following message (where `nnnnn` is the line number causing the break):

```
Break in nnnnn
```

The `STOP` statement differs from the `END` statement since the `STOP` statement leaves all files open.

BASIC always returns control to the command level when a `STOP` statement executes. You may resume exection by giving the `CONT` command.

Example:

```
10 INPUT A,B,C
20 K = A^2 * 5.3 : L = B^3 / .26
30 STOP
40 M = C * K + 100 : PRINT M
RUN
? 1,2,3 [Return]
Break in 30
Ok
PRINT L [Return]
 30.76923
Ok
CONT [Return]
 115.9
Ok
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the interpretive and compiled version of this statement.

STR$ Function

Format: `STR$(x)`

Action: Returns a string representation of the value of *x*.

Also see the `VAL` function.

Example:

```
10 INPUT "ENTER X", X
20 PRINT STR$(X)
RUN
ENTER X45 [Return]
 45
Ok
```

STRING$ Function

Format: `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 an integer between 0 and 255.

Example:

```
10 REM THE ASCII CODE FOR THE DASH SYMBOL IS 45
20 X$ = STRING$(10,45)
30 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

SWAP Statement

Format: `SWAP` *variable1*, *variable2*

Purpose: Exchanges the values of two variables.

Remarks: *variable1* and *variable2* are the identifiers for two variables or array elements.

You may `SWAP` variables of any type (integer, single precision, double precision, or string) as long as both variables are of the same type. If the types for the variables differ, a `Type mismatch` error occurs.

Example:

```
10 A$ = " ONE" : B$ = " ALL" : C$ = " FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

SYSTEM Command/Statement

Format: `SYSTEM`

Purpose: Leaves the BASIC environment and returns control to the operating system.

Remarks: The `SYSTEM` command closes all files and reloads the MS-DOS operating system without deleting any programs or memory except BASIC itself.

You may enter this statement as a Direct Mode command or you may include it as a program statement. For example, if you called BASIC through a Batch file from MS-DOS, the `SYSTEM` command returns control to the Batch file. The Batch file then continues its execution from the point where it left off.

NOTE

Simultaneously pressing the [CTRL] and [C] keys always returns you to the BASIC command level.

NOTE

The BASIC compiler offers no support for this command.

TAB Function

Format: `TAB(j)`

Action: Spaces to the *jth* position on the line. If the current print position is beyond space *j*, `TAB` proceeds to that position on the next line.

Values for *j* may range between 1 and 255. 1 is the leftmost position on a line; the rightmost position is the width minus one.

When *j* is negative, `TAB` treats it as the first character position (that is, *j* = 1).

When *j* is greater than 255, `TAB` rounds the value then calculates the value of `J MOD 256`. `TAB` uses the resulting value.

You may only use the `TAB` statement with either the `PRINT` or `LPRINT` statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "MALLORY ALLISON", "$25.00"
RUN
NAME                    AMOUNT

MALLORY ALLISON         $25.00
Ok
```

TAN Function

Format: `TAN(x)`

Action: Returns the tangent of *x*, where *x* is given in radians.

To convert degrees to radians, multiply the angle by PI/180, where PI = 3.141593.

BASIC evaluates `TAN(X)` with single-precision arithmetic. If the calculation overflows, BASIC displays the `Overflow` error message, sets the result to machine infinity with the appropriate sign, and continues execution.

Example:

```
PRINT TAN(2.22)
-1.317612
```

TIME$ Function

Format: `TIME$`

Action: Retrieves the current time.

The `TIME$` function returns an eight-character string in the form:

hh:mm:ss

where:

hh is the hour of the day, based upon a 24-hour clock. Values range from 00 to 23.

mm is the number of minutes. Values range from 00 to 59.

ss is the number of seconds. Values range from 00 to 59.

Example: This example assumes that the current time is 8:45 P.M.:

```
PRINT TIME$
20:45:00
```

TIME$ Statement

Format: `TIME$ = `*string*

Action: Sets the time for subsequent use by the `TIME$` function.

string represents the current time. It may take one of the following forms:

hh — Sets the hour. (Values may range from 0 to 23.) BASIC sets both minutes and seconds to 00.

hh:mm — Sets both hour and minutes. (Values for minutes may range from 0 to 59.) BASIC sets seconds to 00.

hh:mm:ss — Sets hour, minutes, and seconds. (Values for seconds may range from 0 to 59).

Since the computer uses a 24 hour clock, you must add 12 hours to all times after 12 noon. For example, 8:00 P.M. is 20:00.

Example:

```
TIME$ = "14:"
Ok
PRINT TIME$
14:00:07
Ok
TIME$ = "14:34:04"
Ok
PRINT TIME$
14:34:10
Ok
```

TRON/TROFF Statements

Format: `TRON`

`TROFF`

Purpose: Traces the execution of program statements.

Remarks: You may use the `TRON` statement as a debugging aid in either Direct or Indirect Mode.

The `TRON` statement enables a trace flag. Once set, the trace prints each line number (surrounded by square brackets) when BASIC executes that line.

You can disable the trace flag by giving either a `TROFF` statement or a `NEW` command.

Example:

```
TRON
Ok
10 K = 10
20 FOR J = 1 TO 2
30      L = K + 10
40      PRINT J;K;L
50      K = K + 10
60 NEXT
70 END
RUN
[10][20][30][40]  1     10     20
[50][60][30][40]  2     20     30
[50][60][70]
Ok
TROFF
Ok
RUN
 1     10     20
 2     20     30
Ok
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences in the implementation of these statements.

USR Function

Format: `USR` [*digit*] [`(`*argument*`)`]

Action: Calls an assembly-language subroutine.

digit specifies which `USR` function routine is being called. *digit* may range between 0 and 9 and corresponds to the digit you gave the function with the `DEF USR` statement for that routine.

When you omit *digit*, BASIC assumes `USR0`. See `DEF USR` for further details.

argument is the value you are passing to the subroutine. It may be any numeric or string expression.

In this implementation, if you use a segment other than the default Data Segment (DS), you must execute a `DEF SEG` statement before giving a `USR` function call. The address given in the `DEF SEG` statement determines the address of the subroutine.

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

Example:

```
100 DEF SEG = &HF000
110 DEF USR0 = 0
120 X = Y
130 Y = USR0(X)
140 PRINT Y
```

VAL Function

Format: `VAL(x$)`

Action: Returns the numeric value for the string *x$*. For example, evaluating the following function gives a result of -3:

```
VAL("-3")
```

The `VAL` function strips leading blanks, tabs, and line feed characters from the argument string.

Example: In the following program, lines 20 and 30 show how you may format an `IF` statement by using the line feed character (Control-J).

```
10 READ FIRST$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699
      THEN PRINT FIRST$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) >= 90801 AND VAL(ZIP$) < 90815
      THEN PRINT FIRST$ TAB(25) "LONG BEACH"
40 DATA MARY, CORVALLIS, OREGON, 97330
```

VARPTR Function

Format: VARPTR(*variable*)

VARPTR(#*filenum*)

Action: *filenum* is the number associated with a currently opened file.

variable is a string expression associated with a variable.

When using the *variable* format, the command returns the address of the first byte of data identified with *variable*.

You must assign a value to *variable* before you use it as an argument to VARPTR. Failing to follow this procedure results in an `Illegal function call`.

You may use a variable name of any type (numeric, string, or array).

You normally use VARPTR to obtain the address of a variable or an array so you may pass the address to an assembly-language subroutine.

When passing an array, the best procedure is to pass the lowest-addressed element of that array. Therefore, you should make the function call in the following form when accessing arrays:

```
VARPTR(A(0))
```

For string variables, VARPTR returns the first byte of the string descriptor.

NOTE

You should assign all simple variables before you use VARPTR with an array argument. This is a safeguard since array addresses change whenever you assign a new simple variable.

If you use the *filenum* option, VARPTR returns the starting address of the disc I/O buffer assigned to *filenum*. For random files, VARPTR returns the address of the FIELD buffer assigned to *filenum*.

For either format, the function returns a number that ranges between 0 and 65535. This number is the required offset into the BASIC's Data Segment (DS).

Example:

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

WAIT Statement

Format: `WAIT` *port*, *i* [,*j*]

Purpose: Suspends program execution while monitoring the status of a machine input port.

Remarks: *port* is a port number, which may range from 0 to 65535.

NOTE

This port is a microprocessor port; not one of your computer's datacomm (or peripheral) ports.

---

i and *j* are integer expressions that may range from 0 to 255.

The `WAIT` statement suspends program execution until the specified machine input port develops a specified bit pattern. The data read at the port is `XOR`'ed with the integer expression *j* , and then `AND`ed with *i*. When the result is zero, BASIC loops back and reads the data at the port again. When the result is not zero, execution continues with the next statement.

CAUTION

You could possibly enter an infinite loop when using the `WAIT` statement. To avoid this situation, you must ensure that the specified value appears at the port sometime during program execution. If the program enters an infinite loop, you may exit the loop by simultaneously pressing the [CTRL] and [C] keys.

Example: This example suspends program execution until port 32 receives a 1 bit in the second bit position:

```
100 WAIT 32, 2
```

WHILE. . .WEND Statement

Format: WHILE *expression*

. . .

[*loop statements*]

. . .

WEND

Purpose: Loops through a series of statements as long as the given condition is true.

Remarks: *expression* is a numeric expression which BASIC evaluates. If it is true (not zero), BASIC executes the *loop statements* until it encounters WEND. BASIC then returns to the WHILE statement and checks *expression*. If it is still true, BASIC repeats the entire process. When the expression becomes false, BASIC resumes execution with the statement that follows the WEND statement.

You may nest WHILE/WEND loops to any level. Each WEND matches the most recently encountered WHILE. An unmatched WHILE statement causes a WHILE without WEND error. An unmatched WEND statement causes a WEND without WHILE error.

If you are directing program control to a WHILE loop, you should always enter the loop through the WHILE statement.

Example:

```
10 OPTION BASE 1
20 DIM A(10)
30 REM ---------GET DATA----------
40 DATA 3,2,4,1,5,8,7,6,9,0
50 FOR I = 1 TO 10
60     READ A(I)
70     PRINT A(I);
80 NEXT I
90 REM -------BUBBLE SORT---------
100 J = 10
110 FLIPS = 1       'FORCE ONE PASS THRU LOOP
120 WHILE FLIPS
130    FLIPS = 0
140    FOR I = 1 TO J-1
150        IF A(I) <= A(I+1) THEN 170
160        SWAP A(I), A(I+1) : FLIPS = 1
170    NEXT I
180 WEND
190 PRINT
200 FOR I = 1 TO 10 : PRINT A(I); : NEXT I
RUN
 3  2  4  1  5  8  7  6  9  0
 0  1  2  3  4  5  6  7  8  9
Ok
```

NOTE

If you plan to compile your program, see the BASIC compiler manual for differences between the compiled and interpretive version of this statement.

WIDTH Statement

Format: `WIDTH [LPRINT]` *size*

Purpose: Sets the line width in number of printed characters for the computer screen or a printer.

Remarks: *size* is a numeric expression that may range between 0 and 255. It gives the maximum number of characters that BASIC prints on a logical line. The default setting is 80 characters.

A *size* setting of 255 gives an "infinite" line width. (That is, BASIC never inserts a carriage return character.) Both the `POS` and `LPOS` functions return 0 after the 255th character is printed on a line.

Including the `LPRINT` option sets the line width at the line printer. Omitting this option sets the line width for your computer's screen.

Example:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 13
Ok
RUN
ABCDEFGHIJKLM
NOPQRSTUVWXYZ
Ok
```

NOTE

If you plan to compile your program, check the BASIC compiler manual for differences between the interpretive and compiled versions of this statement.

WRITE Statement

Format: `WRITE` [*list.of.expressions*]

Purpose: Copies data to the computer's screen.

Remarks: *list.of.expressions* is a list of numeric and/or string expressions. You must separate the different items in the list with commas or semicolons.

When you include *list.of.expressions*, BASIC prints the values for the expressions on the computer screen.

Omitting *list.of.expressions* prints a blank line on the screen.

When it prints the line of values, BASIC separates each item from the last with a comma. After it prints the last item in the list, BASIC inserts a carriage return/line feed. BASIC prints quotation marks around any strings within the list.

The `WRITE` statement prints 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# Statement

Format: `WRITE#` *filenum*, *list.of.expressions*

Purpose: Writes data to a sequential disc file.

Remarks: *filenum* is the number you gave the file when you opened it in `O` mode.

list.of.expressions may contain numeric or string expressions or both. You must separate the items in the list with commas or semicolons.

The `WRITE#` statement differs from the `PRINT#` statement by the way it writes data to disc.

`WRITE#` inserts commas between the items as it writes them to disc and surrounds strings with quotation marks. Therefore, you may omit putting explicit delimiters in the list. BASIC inserts a carriage return/line feed character after it writes the last item in the list to disc.

Example: Let `A$ = "CAMERA"` and `B$ = "93604-1"` then the statement:

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

writes the following image to disc:

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

A subsequent `INPUT#` statement, such as:

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

assigns `"CAMERA"` to `A$` and `"93604-1"` to `B$`.

Appendix A

ERROR CODES AND ERROR MESSAGES

This appendix lists the BASIC error messages and describes each one.

Code	Number	Message
NF	1	`NEXT without FOR` A variable in a `NEXT` statement does not correspond to any previously executed, unmatched `FOR` statement variable.
SN	2	`Syntax error` A line is encountered that contains some incorrect sequence of characters (such as a misspelled command, unmatched parentheses, or incorrect punctuation).
RG	3	`RETURN without GOSUB` BASIC encounters a `RETURN` statement for which no previous, unmatched `GOSUB` statement exists.
OD	4	`Out of DATA` BASIC is executing a `READ` statement but no data remains to be read from any `DATA` statement.

Code	Number	Message
FC	5	`Illegal function call` You are attempting to pass a parameter that is out of the permissible range to either a string or mathematical function. This error message also appears under these circumstances: 1. a negative or extremely large subscript 2. a negative or zero argument to `LOG` 3. a negative argument to `SQR` 4. a negative mantissa with a non-integer exponent 5. a call to an `USR` function for which no starting address exists. 6. an improper argument to `MID$`, `LEFT$`, `RIGHT$`, `PEEK`, `POKE`, `TAB`, `SPC`, `STRING$`, `SPACE$`, `INSTR`, or `ON...GOTO`
OV	6	`Overflow` The result of a calculation is too large to be represented in BASIC's number format. When underflow occurs, BASIC sets the result to zero and continues execution.
OM	7	`Out of memory` A program is too large, has too many `FOR` loops or `GOSUBs`, has too many variables, or too many complicated expressions.
UL	8	`Undefined line number` A line referenced in a `GOTO`, `GOSUB`, `IF...THEN...ELSE`, or `DELETE` statement is to a nonexistent line.
BS	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.

Code	Number	Message
DD	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.
/0	11	`Division by zero` BASIC has either encountered a division by zero within an expression or is trying to raise zero to a negative power in an exponentiation. For division by zero, BASIC sets the result to machine infinity with the sign of the numerator. For involution, BASIC sets the result to positive machine infinity. In both cases, execution continues.
ID	12	`Illegal direct` You have attempted to enter a command that is illegal in Direct Mode.
TM	13	`Type mismatch` A string variable name is assigned a numeric value or vice versa. Otherwise, a function that expects a numeric argument is given a string argument or vice versa.
OS	14	`Out of string space` String variables have caused BASIC to exceed the amount of free memory remaining. BASIC allocates string space dynamically, until it runs out of memory.
LS	15	`String too long` An attempt is made to create a string more than 255 characters long.
ST	16	`String formula too complex` A string expression is too long or too complex. You should break the expression into smaller expressions.

Code	Number	Message
CN	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
		3. does not exist
UF	18	Undefined user function
		A USR function is called before the function definition (DEF statement) is given.

The following error messages have no error codes.

19	No RESUME	
	An error-trapping rountine is entered that contains no RESUME statement.	
20	RESUME without error	
	A RESUME statement is encountered before an error-trapping routine is entered.	
21	Unprintable error	
	No error message exists for the detected error condition. This usually results from an ERROR statement 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.	
24-25	Unprintable error	
	No error message exists for the detected error condition. This usually results from an ERROR statement with an undefined error code.	
26	FOR without NEXT	
	A FOR was encountered without a matching NEXT.	

Code	Number	Message
	27-28	`Unprintable error` No error message exists for the detected error condition. This usually results from an `ERROR` statement with an undefined error code.
	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`.
	31-49	`Unprintable error` No error message exists for the detected error condition. This usually results from an `ERROR` statement with an undefined error code.
	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. Report to your Hewlett-Packard service office the conditions under which the message appeared.
	52	`Bad file number` A command references a file with a file number that is not opened or is beyond 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`.

Code	Number	Message
	55	`File already open`
		A sequential output mode `OPEN` is issued for a file that is already open; or a `KILL` is given for an opened file.
	56	`Unprintable error`
		No error message exists for the detected error condition. This usually results from an `ERROR` statement with an undefined error code.
	57	`Device I/O error`
		An I/O error occurred on an I/O operation. It is a fatal error since the operating system cannot recover from this error.
	58	`File already exists`
		The filename specified in a `NAME` statement is identical to a filename already in use on the disc.
	59-60	`Unprintable error`
		No error message exists for the detected error condition. This usually results from an `ERROR` statement with an undefined error code.
	61	`Disk 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. Using `EOF` to detect the end of file avoids this error.
	63	`Bad record number`
		In a `PUT` or `GET` statement, the record number is either greater than the maximum allowed (32767) or is equal to zero.
	64	`Bad file name`
		An illegal form is used for the filename with `LOAD`, `SAVE`, `KILL`, or `OPEN`. (For example, the filename may contain too many characters.)

Code	Number	Message
	65	`Unprintable error`
		No error message exists for the detected error condition. This usually results from an `ERROR` statement with an undefined error code.
	66	`Direct statement in file`
		A Direct Mode statement is encountered while loading an ASCII-formatted 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.
	70	`Disk write protected`
		Your disc has a write protect tab or is a disc that cannot be written to.
	71	`Disk not Ready`
		You have probably inserted the disc improperly.
	72	`Disk media error`
		A hardware or disc problem occurred while the disc was being written to or read from. (For example, the disc drive may be malfunctioning or the disc may be damaged.)
	74	`Rename across disks`
		An attempt was made to rename a file with a new drive destination. As this is not allowed, the operation is canceled.

Appendix B

USING TERMINAL FEATURES IN BASIC

Introduction

You can program the terminal portion of your computer to perform many of the functions of an intelligent terminal. By using these features, you can tell the computer to perform tasks that would otherwise be done within each application program.

Most tasks that you do at the keyboard can also be done under program control with escape sequences. An escape sequence is simply a series of ASCII characters preceded by the escape character, ESC (ASCII code 27). Each escape sequence tells the computer to do a certain task. For example, the escape sequence `ESC h` "homes" the cursor to the upper left-hand corner of the screen.

This appendix shows some examples of how you might use escape sequences. For a list of all the escape sequences that you can use on your Portable PLUS, refer to the *Portable PLUS Technical Reference Manual* (HP 45559K), which is available from your HP sales representative.

NOTE

For clarity, this appendix shows a space between each character in an escape sequence. When you type in the sequence, do NOT insert spaces.

Escape sequences fall into two categories: two-character sequences and multiple-character sequences. For two-character sequences, you must press the keys in order and use the correct case (upper- or lower-case). For example, `ESC B` ([Esc] then the shifted [B] key) moves the cursor down one row whereas `ESC b` ([ESC] then the unshifted [B] key) unlocks the keyboard. The difference appears subtle but is quite important to your computer.

Multiple-character escape sequences have one or more groups of characters. Each group, consisting of a number or other character followed by a letter, specifies one parameter of the sequence. Generally, you can arrange these groups in any order or even leave some out entirely, depending on the task you want your computer to do. In this type of escape sequence, a capital letter defines the end of the escape sequence, so you would capitalize only the last letter and type the rest in lower case. An example is the cursor-positioning escape sequence. Here is an escape sequence that positions the cursor at row zero, column zero ("home"):

```
ESC & a 0 r 0 C
```

The uppercase `C` ends the sequence. But since you can interchange the order of groups in a multiple-character escape sequence, you could also use:

```
ESC & a 0 c 0 R
```

This escape sequence does the same task as the other one. Only the order of groups of characters is different. Again, the capital letter ended the sequence. The order of the groups of characters is not critical as long as the last character is an uppercase letter.

You may also truncate this command. To position the cursor to the top line, without affecting the column position, use the following sequence:

```
ESC & a 0 R
```

Notice that the `C` or column parameter is simply omitted. The upper case `R` terminates the sequence.

You must be aware of two situations when using escape sequences with BASIC.

1. The `PRINT` statement forces a carriage return/line feed after every `PRINT` statement unless the string to be printed is followed by a semicolon (`;`). If you print a sequence that positions the cursor, and forget to end the `PRINT` statement with a semicolon, the cursor automatically moves to the next line.

2. BASIC monitors the number of characters printed on each line so that a carriage return/line feed can be added after every 80 characters. When you are using `PRINT` statements to generate escape sequences, you may not want these characters added automatically. When you use the `WIDTH` statement with a value of 255, BASIC stops inserting the automatic carriage return/line feed and permits your program to fully utilize terminal control sequences.

Sample Functions

An example of using escape sequences within a BASIC program is illustrated below. By using these sample functions as a model, you should be able to program any of the remaining functions that are described in the HP 150 MS-DOS User's Guide.

The function definitions have been entered on multiple lines just as you see them. If the program lines were entered normally, each line could contain a maximum of 80 characters. This makes it difficult to format the program listing as you see it here. However, by pressing [CTRL] [J] at the end of each line, BASIC allows single line statements to be entered on multiple lines.

```
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 DEF FNKEY$(K,A$,B$) = ESC$ + "&f0a" + STR$(K) + "k" +
     STR$(LEN(A$)) + "d" + STR$(LEN(B$) + 1) + "L" + A$ + B$ +
     CHR$(13)
1050 DEF FNIV$(A$) = ESC$ + "&dB" + A$ + ESC$ + "&d@"
```

Before exploring how these functions might be used within a program, let's take a closer look at each one.

FNHOME$ executes a Home-up, clear-display sequence. This places the cursor at the top of the display and clears the screen (by deleting the contents of display memory).

FNCURSOR$ positions the cursor to the row and column specified by R and C. Note that you must use STR$ to convert the numeric values of C and R into a string representation of the desired values.

FNKEY$ allows you to define any of the User Keys. The key to be defined is specified as K, the label as A$, and the definition as B$. Note that the string representation of the length of each field must be specified. As with the cursor function above, you must convert the numeric value to a string.

FNIV$ prints the string of characters in A$ in inverse video at the current cursor position. FNIV$ also guarantees that only A$ is shown in inverse video by specifically disabling all 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 causes program execution to continue, while one terminates the program. The prompt requesting operator input appears in the center of the display in inverse video.

```
1060 WIDTH 255
1070 PRINT FNHOME$;
1080 PRINT FNKEY$(1,"CONTINUE","PROCEED");
1090 PRINT FNKEY$(8, "EXIT TO MS-DOS", "EXIT");
1100 PRINT ESC$ + "&jB";
1110 PRINT FNCURSOR$(10,20);
1120 PRINT FNIV$("CONTINUE?");
1130 PRINT FNCURSOR$(20,20);
1140 INPUT " ", A$
1150 IF A$ = "PROCEED" GOTO 2000
1160 IF A$ = "EXIT" GOTO 5000
2000 GOTO 1000
5000 STOP
5010 END
```

Remember, the semicolon is used after each `PRINT` statement to allow the programmer to position the cursor wherever necessary. This prevents BASIC from performing an automatic carriage return as it normally would.

For further information on programming with escape sequences, refer to the appropriate sections in the HP 150 MS-DOS User's Guide.

Appendix C

REFERENCE TABLES

ASCII Character Codes

ASCII Code	Character	Description
000	NUL	Null
001	SOH	Start of heading
002	STX	Start of text
003	ETX	End of text
004	EOT	End of transmission
005	ENQ	Enquiry
006	ACK	Acknowledge
007	BEL	Bell
008	BS	Backspace
009	HT	Horizontal tabulation
010	LF	Line feed
011	VT	Vertical tabulation
012	FF	Form feed
013	CR	Carriage return
014	SO	Shift out
015	SI	Shift in
016	DLE	Data Link Escape
017	DC1	Device control 1 or X-ON
018	DC2	Device control 2
019	DC3	Device control 3 or X-OFF
020	DC4	Device control 4
021	NAK	Negative acknowledge
022	SYN	Synchronous idle

ASCII Code	Character	Description
023	ETB	End of transmission block
024	CAN	Cancel
025	EM	End of medium
026	SUB	Substitute
027	ESC	Escape
028	FS	File separator
029	GS	Group separator
030	RS	Record separator
031	US	Unit separator
032	SPACE	Space
033	!	Exclamation point
034	"	Quotation mark
035	#	Number sign (pound sign or hash mark)
036	$	Dollar sign
037	%	Percent sign
038	&	Ampersand
039	'	Apostrophe (closing single quote)
040	(	Opening parenthesis
041	)	Closing parenthesis
042	*	Asterisk
043	+	Plus
044	,	Comma
045	-	Hyphen (minus)
046	.	Period (point)
047	/	Slant (solidus)
048	0	Zero
049	1	
050	2	
051	3	
052	4	
053	5	
054	6	
055	7	
056	8	
057	9	
058	:	Colon
059	;	Semicolon

ASCII Code	Character	Description
060	<	Less than sign
061	=	Equal
062	>	Greater than sign
063	?	Question mark
064	@	Commercial at sign
065	A	
066	B	
067	C	
068	D	
069	E	
070	F	
071	G	
072	H	
073	I	
074	J	
075	K	
076	L	
077	M	
078	N	
079	O	
080	P	
081	Q	
082	R	

ASCII Code	Character	Description
083	S	
084	T	
085	U	
086	V	
087	W	
088	X	
089	Y	
090	Z	
091	[	Opening square bracket
092	\	Back slant
093	]	Closing square bracket
094	^	Caret (upward arrow)
095	__	Underscore
096	‘	Opening single quote
097	a	
098	b	
099	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	

ASCII Code	Character	Description
123	{	Opening brace (curly bracket)
124	|	Vertical line
125	}	Closing brace (curly bracket)
126	~	Tilde
127	DEL	Delete (rub out)
128		Undefined control code
129		Undefined control code
130		Undefined control code
131		Undefined control code
132		Undefined control code
133		Undefined control code
134		Undefined control code
135		Undefined control code
136		Undefined control code
137		Undefined control code
138		Undefined control code
139		Undefined control code
140		Undefined control code
141		Undefined control code
142		Undefined control code
143		Undefined control code
144		Undefined control code
145		Undefined control code
146		Undefined control code
147		Undefined control code
148		Undefined control code
149		Undefined control code

ASCII Code	Character	Description
150		Undefined control code
151		Undefined control code
152		Undefined control code
153		Undefined control code
154		Undefined control code
155		Undefined control code
156		Undefined control code
157		Undefined control code
158		Undefined control code
159		Undefined control code
160		Do not use
161	À	Uppercase A accent grave
162	Â	Uppercase A circumflex
163	È	Uppercase E accent grave
164	Ê	Uppercase E circumflex
165	Ë	Uppercase E umlaut or diaeresis
166	Î	Uppercase I circumflex
167	Ï	Uppercase I umlaut or diaeresis
168	´	Accent acute
169	`	Accent grave
170	^	Circumflex accent
171	¨	Umlaut (diaeresis) accent
172	~	Tilde accent
173	Ù	Uppercase U accent grave
174	Û	Uppercase U circumflex
175	₤	Italian lira symbol
176	‾	Overline (high line)
177		Undefined control code
178		Undefined control code
179	°	Degree (ring)
180	Ç	Uppercase C cedilla
181	ç	Lowercase c cedilla
182	Ñ	Uppercase N tilde
183	ñ	Lowercase n tilde
184	¡	Inverse exclamation mark
185	¿	Inverse question mark
186	¤	General currency symbol
187	£	British pound sign
188	¥	Japanese yen symbol
189	§	Section sign

ASCII Code	Character	Description
190	ƒ	Dutch guilder symbol
191	¢	Cent sign
192	â	Lowercase a circumflex
193	ê	Lowercase e circumflex
194	ô	Lowercase o circumflex
195	û	Lowercase u circumflex
196	á	Lowercase a accent acute
197	é	Lowercase e accent acute
198	ó	Lowercase o accent acute
199	ú	Lowercase u accent acute
200	à	Lowercase a accent grave
201	è	Lowercase e accent grave
202	ò	Lowercase o accent grave
203	ù	Lowercase u accent grave
204	ä	Lowercase a umlaut or diaeresis
205	ë	Lowercase e umlaut or diaeresis
206	ö	Lowercase o umlaut or diaeresis
207	ü	Lowercase u umlaut or diaeresis
208	Å	Uppercase A degree
209	î	Lowercase i circumflex
210	Ø	Uppercase O crossbar
211	Æ	Uppercase AE ligature
212	å	Lowercase a degree
213	í	Lowercase i accent acute
214	ø	Lowercase o crossbar
215	æ	Lowercase ae ligature
216	Ä	Uppercase A umlaut or diaeresis
217	ì	Lowercase i accent grave
218	Ö	Uppercase O umlaut or diaeresis
219	Ü	Uppercase U umlaut or diaeresis
220	É	Uppercase E accent acute
221	ï	Lowercase i umlaut or diaeresis
222	ß	Sharp s
223	Ô	Uppercase O circumflex
224	Á	Uppercase A accent acute
225	Ã	Uppercase A tilde
226	ã	Lowercase a tilde
227	Đ	Uppercase D with stroke
228	đ	Lowercase d with stroke
229	Í	Uppercase I accent acute

ASCII Code	Character	Description
230	Ì	Uppercase I accent grave
231	Ó	Uppercase O accent acute
232	Ò	Uppercase O accent grave
233	Õ	Uppercase O tilde
234	õ	Lowercase o tilde
235	Š	Uppercase S with caron
236	š	Lowercase s with caron
237	Ú	Uppercase U accent acute
238	Ÿ	Uppercase Y umlaut or diaeresis
239	ÿ	Lowercase y umlaut or diaeresis
240	Þ	Uppercase thorn
241	þ	Lowercase thorn
242		Undefined
243		Undefined
244		Undefined
245		Undefined
246	—	Long dash (horizontal bar)
247	¼	One fourth
248	½	One half
249	ª	Feminine ordinal indicator
250	º	Masculine ordinal indicator
251	«	Opening guillemets (angle quotes)
252	■	Solid
253	»	Closing guillemets (angle quotes)
254	±	Plus/minus sign
255		Do not use

ROMAN8 CHARACTER SET
(USASCII PLUS ROMAN EXTENSION)

				b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
				b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
				b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
				b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b4	b3	b2	b1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p				¯	â	Å	Á	Þ
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q			À		ê	Î	Ã	þ
0	0	1	0	2	STX	DC2	"	2	B	R	b	r			Â		ô	Ø	ã	
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s			È	°	û	Æ	Ð	
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t			Ê	Ç	á	å	đ	
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u			Ë	ç	é	í	Í	
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v			Î	Ñ	ó	ø	Ì	—
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w			Ï	ñ	ú	æ	Ó	¼
1	0	0	0	8	BS	CAN	(	8	H	X	h	x			´	¡	à	Ä	Ò	½
1	0	0	1	9	HT	EM	)	9	I	Y	i	y			`	¿	è	ì	Õ	ª
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z			^	¤	ò	Ö	õ	º
1	0	1	1	11	VT	ESC	+	;	K	[	k	{			¨	£	ù	Ü	Š	«
1	1	0	0	12	FF	FS	,	<	L	\	l	|			˜	¥	ä	É	š	■
1	1	0	1	13	CR	GS	-	=	M	]	m	}			Ù	§	ë	ï	Ú	»
1	1	1	0	14	SO	RS	.	>	N	^	n	~			Û	ƒ	ö	ß	Ÿ	±
1	1	1	1	15	SI	US	/	?	O	_	o	DEL			₤	¢	ü	Ô	ÿ	

Reserved Words

The following table lists all the reserved words in BASIC.

ABS	ERASE	LPOS	RND
AND	ERL	LPRINT	RSET
ASC	ERR	LSET	RUN
ATN	ERROR	MERGE	SAVE
AUTO	EXP	MID$	SGN
BLOAD	FIELD	MKD$	SIN
BSAVE	FILES	MKI$	SPACE$
CALL	FIX	MKS$	SPC
CDBL	FN*xxxxxxxx*	MOD	SQR
CHAIN	FOR	NAME	STEP
CHR$	FRE	NEW	STOP
CINT	GET	NEXT	STR$
CLEAR	GOSUB	NOT	STRING$
CLOSE	GOTO	OCT$	SWAP
COMMON	HEX$	OFF	SYSTEM
CONT	IF	ON	TAB
COS	IMP	OPEN	TAN
CSNG	INKEY$	OPTION	THEN
CVD	INP	OR	TIME$
CVI	INPUT	OUT	TO
CVS	INPUT#	PEEK	TROFF
DATA	INPUT$	POKE	TRON
DATE$	INSTR	POS	USING
DEF	INT	PRINT	USR
DEFDBL	KILL	PRINT#	VAL
DEFINT	LEFT$	PUT	VARPTR
DEFSNG	LEN	RANDOMIZE	WAIT
DEFSTR	LET	READ	WEND
DELETE	LINE	REM	WHILE
DIM	LIST	RENUM	WIDTH
EDIT	LLIST	RESET	WRITE
ELSE	LOAD	RESTORE	WRITE#
END	LOC	RESUME	XOR
EOF	LOF	RETURN	
EQV	LOG	RIGHT$	

Appendix D

ASSEMBLY LANGUAGE SUBROUTINES

Introduction

This appendix is provided for users who call assembly-language subroutines from their BASIC programs. If you do not use assembly-language subroutines, you may omit reading this appendix.

The `USR` function allows assembly-language subroutines to be called in the same way that BASIC intrinsic functions are called. However, we recommend that you use the `CALL` or `CALLS` statement for interfacing machine-language programs with BASIC. These statements produce more readable source code and can pass multiple arguments. In addition, the `CALL` statement is compatible with more languages than the `USR` function.

Memory Allocation

You must set aside memory space for an assembly-language subroutine before you can load it. You accomplish this through the `/M:` switch in the BASIC command line. (The `/M:` switch sets the highest memory location that BASIC uses.)

In addition to the BASIC Interpreter code area, BASIC uses up to 64K of memory beginning at the Data Segment (DS).

When calling an assembly-language subroutine, if you need more stack space, you can save the BASIC stack and set up a new stack for the assembly-language subroutine. You must restore the BASIC stack, however, before the program returns from the subroutine.

You can load an assembly-language subroutine into memory through the operating system or the `POKE` statement. If you have the software package for your microprocessor, routines may be assembled with the MACRO Assembler and linked, but not loaded, using the LINK Linking Loader. To load the program file, observe these guidelines:

- Make sure the subroutines do not contain any long references
- Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file

CALL Statement

The `CALL` statement is the recommended way of interfacing machine-language programs with BASIC. Do not use the `USR` function unless you are running previously written programs that already contain `USR` functions.

Format: `CALL` *variable.name* [`(`*argument.list*`)`]

Remarks: *variable.name* contains the segment offset that is the starting point in memory of the subroutine that you are calling.

argument.list contains the variables or constants that are passed to the routine. You must separate the items in the list with commas.

Invoking the `CALL` statement causes the following events:

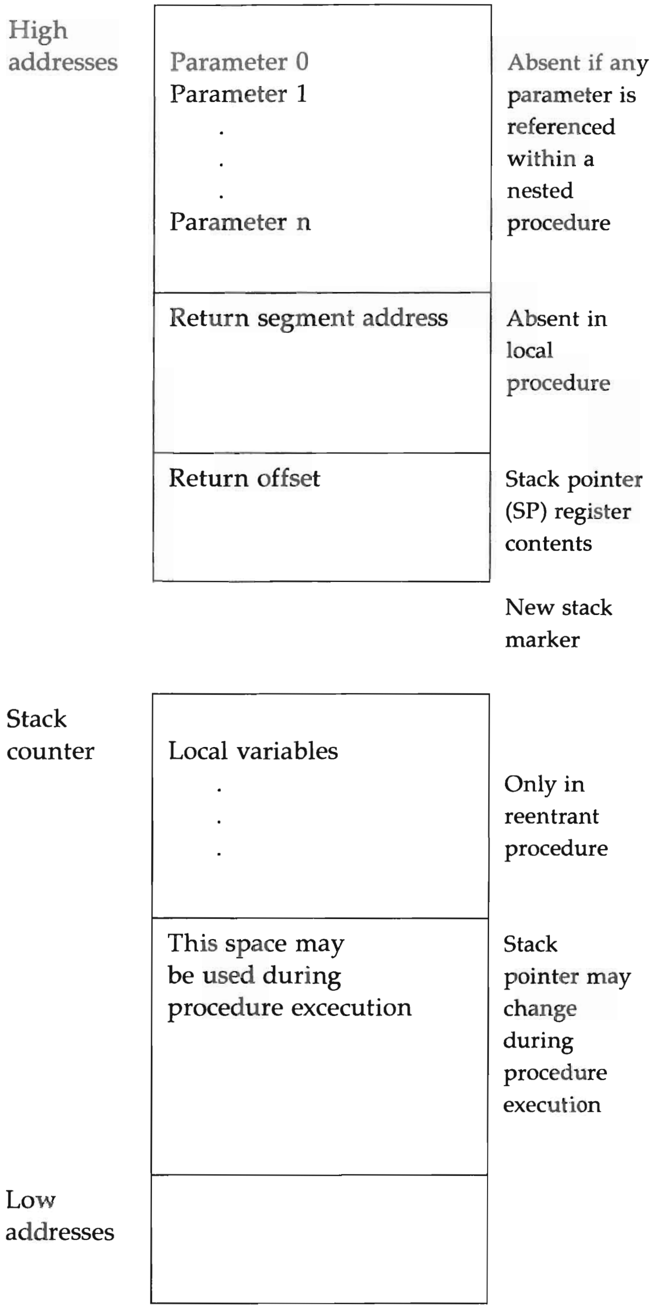
- For each parameter in the argument list, the 2-byte offset of the parameter's location within the Data Segment (DS) is pushed onto the stack.
- The BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.
- Control is transferred to your routine through a long call to the segment address given in the last `DEF SEG` statement and the offset given in *variable.name*.

The following table illustrates the state of the stack at the time the `CALL` statement executes.

High addresses	Parameter 0 Parameter 1 . . . Parameter n	Each parameter is a 2-byte pointer into memory
Stack counter	Return segment address	
	Return offset	
Low addresses		Stack pointer (SP) register contents

Your routine now has control. You may refer to parameters by moving the stack pointer to the base pointer, then adding a positive offset to the base pointer.

The following figure shows the condition of the stack during execution of the called subroutine.



The following rules apply when coding a subroutine:

1. The called routine may destroy the AX, BX, CX, DX, SI, and DI registers.

2. The called program must know the number and length of the parameters passed. References to parameters are positive offsets to BP (assuming the called routine moved the current stack pointer into BP).

3. The called routine must do a RET *n* statement, where *n* is twice the number of parameters in the argument list. This statement adjusts the stack to the start of the calling sequence.

4. Values are returned to BASIC by including a variable name in the argument list to receive the result.

5. If the argument is a string, the parameter's offset points to three bytes, which, as a unit, is called the **string descriptor**.

Byte 0 of the string descriptor contains the length of the string. This number may vary from 0 (if all 8 bits are zero) to 255 (if all 8 bits are ones).

Bytes 1 and 2, respectively, are the lower and upper 8 bits of the starting string address in string space.

CAUTION

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add `+" "` to the string literal in the program. For example, the following statement forces the string literal to be copied into string space:

```
20 A$ = "BASIC" + " "
```

You may now modify this string without affecting the program.

---

6. Strings may be altered by user routines, but their length MUST REMAIN THE SAME. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example:

```
100 DEF SEG = &H800
110 FOO = &H7A
120 CALL FOO(A,B$,C)
```

Line 100 sets the segment address to 8000 Hex. The value of the variable FOO is added to the address as an offset to the DEF SEG segment value. (See a book on 8086/8088 microprocessors for a complete discussion of segment addressing.) Here FOO is set to &H7FA, so that the call to FOO executes the subroutine at location 8000:7FA Hex (equivalent to absolute address 807FA).

The following sequence in assembly-language code demonstrates access of the parameters passed. The return result is stored in variable "C".

```
PUSH BP          ;Save BP register
MOV BP,SP        ;Get current stack position in BP
MOV BX,[BP+8]    ;Get address of B$ dope
MOV CL,[BX]      ;Get length of B$ in CL
MOV DX,[BX+1]    ;Get address of B$ text in DX
        .
        .
        .
MOV SI,[BP+10]   ;Get address of 'A' in SI
MOV DI,[BP+6]    ;Get pointer to 'C' in DI
MOVS WORD        ;Store variable 'A' in 'C'.
POP BP           ;Restore BP register
RET 6            ;Restore stack, return
```

NOTE

The called program must know the variable type for the numeric parameters passed. In the previous example, the instruction MOVS WORD copies only 2 bytes. This suffices when variables A and C are integers. However, you have to copy 4 bytes if the variables are single-precision values and 8 bytes if they are double-precision values.

USR Function

Although the `CALL` statement is the recommended way of calling assembly-language subroutines, the `USR` function is still available for compatibility with previously written programs.

Format: `USR` [*digit*] (*argument*)

Remarks: *digit* is an integer that ranges from 0 to 9. It specifies which `USR` routine is being called and corresponds with the digit supplied in the `DEF USR` statement for that routine. If you omit *digit*, BASIC assumes the call is to `USR0`.

argument is any numeric or string expression.

In BASIC, you must execute a `DEF USR` statment before calling a `USR` function to ensure that the code segment points to the subroutine being called. The address given in the `DEF SEG` statement determines the starting address of the subroutine.

For each `USR` function, you must execute a `DEF USR` statement to define the `USR` function offset. This offset and the currently active `DEF SEG` statement determines the starting segment of the subroutine.

When the `USR` function call is made, register AL contains a value that specifies which type of argument was given. The value in AL may be one of the following:

Value in AL	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, the BX register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

The Floating Point Accumulator is the exponent minus 128. (The radix point is to the left of the most significant bit of the mantissa.)

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.
FAC-3 contains the lower 8 bits of the argument.

If the argument is a single-precision floating point number:

FAC-2 contains the middle 8 bits of the argument.
FAC-3 contains the lowest 8 bits of the argument.

If the argument is a double-precision floating point number:

FAC-7 through FAC-4 contain four more bytes of the mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register pair points to three bytes. These three bytes are called the **string descriptor**.

Byte 0 contains the length of the string. This value varies from 0 (if all 8 bits are zeros) to 255 (if all 8 bits are ones).

Bytes 1 and 2, respectively, are the lower and upper eight bits of the starting string address in the BASIC Data Segment.

CAUTION

If the argument is a string literal in the program, the string descriptor points to program text. Be careful not to alter or destroy your program this way.

Usually, the value returned by a USR function is the same type (integer, single-precision, double-precision, or string) as the argument that was passed to it.

Example:

```
100 DEF USR0=&H800 'Assumes user gave /M:32767
120 X = 5
130 Y = USR0
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with the argument passed.

Appendix E

INSTALLING BASIC ON THE HP 110

Introduction

This appendix provides details on installing BASIC on the HP 110 Portable Computer. It tells you how to make a back-up copy of your master disc and the simpliest procedures for getting BASIC up and running on your computer.

You have two major options. You may either modify P.A.M. so you can use P.A.M.'s friendly interface to run BASIC or you may simply enter `BASIC` as an MS-DOS system command. This appendix describes both methods.

Copying The Program Disc For Back-Up

Before using Series 100/BASIC for the first time, you should make a back-up copy of the master BASIC disc. To accomplish this, you need the following:

- the Portable
- the Series 100/BASIC program disc
- an HP 9114A 3 ½-Inch Single Flexible Disc Drive (or another compatible disc drive)
- a back-up disc, formatted as a single-sided disc

CAUTION

Before going through the install procedure, you should write-protect your master disc to prevent any accidental "over-writing". For information on write-protecting your disc, refer to the owner's manual that accompanied your disc drive.

The Portable encorporates many new technologies into its design, including the use of double-sided discs. It is important, however, that the Portable remains compatible with other Hewlett-Packard Series 100 products. Since all existing Series 100 software uses single-sided disc format, you should copy your master BASIC disc as a single-sided disc. This requires your using the format program on the UTILITIES disc that came with your Portable as the Portable's built-in format command formats a disc in double-sided format.

NOTE

The UTILITIES disc is a double-sided disc. This means that you must read it in a double-sided disc drive. If you have a single-sided disc drive, you should use P.A.M. or the MS-DOS `FORMAT` command to format the back-up disc.

Double-sided disc drives (such as the HP 9114A) can use single-sided discs without any problems.

Formatting The Back-Up Disc

If you are using a new disc, you must format it first.

Step 1. Connect and turn on all the equipment. You should also ensure that the Portable's System Configuration menu correctly show the number of disc drives that you have connected to your Portable. (The HP 110 Portable Computer Owner's Manual provides the necessary details.)

Step 2. Insert the UTILITIES disc that you received with your Portable into a double-sided disc drive.

Step 3. To copy the formatting program to your electronic disc, type:

```
COPY C:FORMAT.COM A: [Return]
```

Step 4. Remove the UTILITIES disc from the disc drive and insert the blank, unformatted disc.

Step 5. Type `A:FORMAT C: /W` [Return]

Then press the [Return] key to begin formatting. It takes about two minutes for the system to format the disc.

Step 6. After formatting finishes, you are ready to copy the master disc. But first, erase the formatting program from the electronic disc by typing:

```
ERASE A:FORMAT.COM [Return]
```

Making The Back-Up Copy

Once you have a single-sided, formatted disc, you can use the MS-DOS command `DISKCOPY` to copy the "Source" disc (the Series 100/BASIC program disc) to the formatted "Target" disc (your back-up disc).

The master BASIC disc contains these files:

- `BASIC.COM` (the BASIC interpreter program)
- `BASIC.IN$` (the HP 150 install file)
- `RANDOM.BAS` (a sample BASIC program)
- `PAM.MNU` (the HP 110 P.A.M. menu file)
- `HP110\BASIC.IN$` (an HP 110 file for a future install program)

You must copy all these files to your back-up disc. The Portable provides the necessary prompts to lead you through this process.

If you have a dual disc drive, type:

`DISKCOPY C: D:` [Return]

Now follow the instructions on the display. (They direct you to place the "Source" (your master) disc in drive C and the "Target" (your back-up) disc in drive D).

The procedure for a single disc drive involves a few more steps but the Portable again provides assistance.

Step 1. If you have a single disc drive connected to your system, type:

`DISKCOPY C: C:` [Return]

Step 2. At this point, the Portable prompts you to insert the "Target" disc (your formatted back-up disc) into the disc drive and press any key when you are ready to continue.

NOTE

Your Portable detects the [CTRL], [Shift], and [Extend char] keys as keys that are used in combination with other keys. Therefore, it does not respond to your pressing any of these keys by themselves. Although you may press any other key to continue the operation, the remainder of this procedure directs you to press the [Return] key.

Step 3. The Portable then tells you when to insert the Source disc, when to insert the Target disc again, the Source disc, the Target disc, and so on. To continue the copying process, swap the discs and press the [Return] key. Keep swapping discs in the external drive as the Portable directs until all of the master files are copied. The copying process is done when you see the message `Copy complete`.

Step 4. As soon as the copying is finished, you are asked if you want to make another copy. If you do, press the [Y] key and repeat the above procedure with another formatted back-up disc. If your answer is no, press the [N] key then the [Return] key to return to P.A.M..

Step 5. Once you have Series 100/BASIC on a back-up disc, you should use this back-up disc as your work disc and store the master program disc in a safe place. (When you remove the back-up disc from the disc drive, don't forget to label it for future reference.)

Running Series 100/BASIC

You can load Series 100/BASIC through P.A.M. or directly from the MS-DOS operating system. P.A.M. provides a "friendlier" interface but requires more steps in the set-up procedure. Entering BASIC through an MS-DOS system command gives you more flexibility in establishing the BASIC environment (see Chapter 3 for further information). This appendix uses the simpliest form of the `BASIC` command.

Running BASIC Using P.A.M.

You can use P.A.M. to run Series 100/BASIC from either an external disc drive or the internal electronic disc.

Running From An External Disc

Step 1. Display the main P.A.M. menu on your screen. If some other information currently appears, you can return to P.A.M. by entering the MS-DOS `EXIT` command, or by performing a hard reset. (You may reset your Portable by simultaneously pressing the [Shift] [CTRL] and [Break] keys.)

Step 2. Place your back-up copy of the BASIC disc into the external disc drive.

Step 3. Press [f4] (Reread Discs).

This action updates the P.A.M. menu to include the BASIC label as a possible selection.

Step 4. Use the [Tab] key or the cursor-control ("arrow") keys to move the pointer to the `BASIC` field.

Step 5. Press [Select] or [f1] (Start Applic).

Running From The Electronic Disc

Before you can use P.A.M.'s facilities to run Series 100/BASIC, you must copy the `BASIC.COM` file into the electronic disc. Next, you must install the program in P.A.M. by modifying the `PAM.MNU` file in the electronic disc. You do this by placing two lines into the existing `PAM.MNU` file to reserve space for the label and file name. These lines are:

- `Basic`
- `BASIC`

If the electronic disc doesn't have a `PAM.MNU` file, you can copy the one from your back-up disc to the electronic disc.

To remove BASIC from P.A.M., you must return the `PAM.MNU` file to its original state. (Since the install procedure added two lines to the `PAM.MNU` file, you must delete those same two lines.)

For information on these tasks, refer to "Copying a File" and "Installing Application Programs in P.A.M." in chapter 2 of your HP 110 Portable Computer Owner's Manual.

Step 1. Display the main P.A.M. menu on your screen. (You can return to the P.A.M. menu by entering the MS-DOS `EXIT` command, or by performing a hard reset.)

Step 2. Use the MS-DOS "list-directory" command (`DIR`) to verify that the `BASIC.COM` file is in the electronic disc.

Step 3. If the application program is installed, use the [Tab] key or the "arrow" keys to move the pointer to the `BASIC` selection.

Step 4. Press [Select] or [f1] (Start Applic).

Running BASIC Using MS-DOS

You can also run Series 100/BASIC from an external disc or the electronic disc by typing the appropriate MS-DOS command. The following discussion gives the simpliest form of the `BASIC` command. Refer to Chapter 3 if you want to tailor the BASIC environment for your specific needs.

Running From An External Disc

Step 1. Insert your back-up copy of the BASIC disc into the external disc drive.

Step 2. Type `C:BASIC` [Return]

When drive C (the drive with the BASIC disc) is the default drive, you may omit typing the drive specifier `C:`.

Running From The Electronic Disc

Before using MS-DOS to run Series 100/BASIC from the electronic disc, you must copy the `BASIC.COM` file into the electronic disc. For information on how to do this, refer to "Copying a File" in chapter 2 of your HP 110 Portable Computer Owner's Manual.

Step 1. Use the MS-DOS "list-directory" command (`DIR`) to verify that the `BASIC.COM` file is in the electronic disc.

Step 2. Type `A:BASIC` [Return]

If drive A is the default drive, you may omit typing the drive specifier `A:`.

Step 3. Remove the SYS__MASTER disc and insert your copy of DISC APPLICATIONS into the flexible disc drive A. (This disc contains the "Install" utility.)

Step 4. Touch `Reread Discs`.

P.A.M. updates the list of possible selections to include the INSTALL utility.

Step 5. Touch the `INSTALL` field to select this option. (You know you have successfully selected Install when the field becomes highlighted.)

Step 6. Touch `Start Applic`.

The message `Loading Install` appears on your screen.

Step 7. When the red light on the disc drive goes out, remove the DISC APPLICATIONS disc and insert your Series 100/BASIC master disc.

Step 8. Inspect the current screen display. The message `Select a function below` appears on the screen. Since you want to install an application program and not remove one, touch `Install Applics`.

Step 9. The next screen asks you to select the correct disc drives. Remember, you are installing BASIC "FROM:" the flexible disc drive A "TO:" the fixed disc drive B.

Step 10. Touch "`A`" in the FROM column until this field is highlighted.

Step 11. Touch "`B`" in the TO column until this field is highlighted.

Step 12. Touch `Show Applics`.

The message `Select the applications to be installed` appears on the screen.

Step 13. Select `BASIC` by touching this field.

Step 14. Touch `Start Install`.

Step 15. When the installation procedure finishes, the message `Install completed` appears on the screen. Touch `Exit Select`.

NOTE

Step 16 applies to the initial version of P.A.M. (version number A.01.02). If you have a later version of P.A.M., proceed to Step 17.

Step 16. Touch `Exit Install`. This is your last step when using version number A.01.02 of P.A.M..

Step 17. Touch `Main Menu`, and after the Main Menu appears touch `Exit Main`.

Appendix F

INSTALLING BASIC ON THE HP 150

Introduction

This appendix provides details on installing BASIC on the HP 150 Personal Computer. It tells you how to make a back-up copy of your master disc and the simplest procedures for getting BASIC up and running on your computer.

Making A Working Copy Of BASIC

You should always make a back-up copy of your application software as a safeguard against possible damage or loss. Since the HP 150 supports a variety of peripheral, mass-storage devices, the actual procedure depends upon which disc drive you are using. The following sections describe making a working copy of BASIC using either a dual disc drive or a hard disc drive. As the system directs you on each step you must take, you may follow the instructions on the screen if you have a different type of disc drive.

CAUTION

Before going through the install procedure, you should write-protect your master disc to prevent any accidental "over-writing". For information on write-protecting your disc, refer to the owner's manual that accompanied your disc drive.

For Dual Disc Drive Users

The following discussion lists the steps that you should follow to make a back-up copy of your BASIC master disc. For this procedure, you need the following discs:

- Your back-up copy of the HP 150 SYS__MASTER
- Your back-up copy of DISC APPLICATIONS
- Your master copy of BASIC
- An unformatted disc

Your computer assumes drive A (the left-hand drive) is the currently active drive, unless you have taken steps to instruct it differently. This procedure, therefore, requires your inserting the "controlling" discs into drive A.

Inserting a disc into a drive is an easy task:

- Hold the disc by its label end to prevent soiling the shutter mechanism.
- Inspect both sides of the disc. You can recognize the top since it has printing on the shutter and also contains the larger portion of the label. The most obvious feature on the bottom is the circular head.
- Ensure that the top of the disc is facing up when you insert the disc into a drive. The engraved arrow shows which way you enter the disc.

The following discussion uses the touch fields of the HP 150, but you may select each operation by pressing the function key that corresponds to the operation you wish to perform.

Step 1. Put your back-up copy of the HP 150 Sys__Master (the one containing "P.A.M.") into drive A.

Step 2. Put the disc you wish to format in drive B.

Step 3. Do a System Reset (by simultaneously pressing the [Shift], [CTRL], and [Reset] keys) to put the system in its initial, power-on state.

Step 4. Select the FORMAT program by touching this field, then touch Start Applic.

Step 5. Select `drive B` by pressing this field and type a label for the disc (for example, `BASIC`) . Then press the [Return] key.

Step 6. To include a copy of P.A.M. on the disc while it is being formatted, touch `Copy System`. (An asterisk appears in the screen label to show that you have selected this option.)

Step 7. Touch `Start Format` to read in P.A.M.. When the message `Press Return to continue` appears, press the [Return] key to format your disc on drive B.

Step 8. After formatting finishes, touch `Exit FORMAT` to leave this application.

Step 9. Remove the back-up copy of P.A.M. from drive A and insert your back-up copy of DISC APPLICATIONS into drive A. (This disc contains the "Install" utility.)

Step 10. Touch `Reread Discs`.

Step 11. Select `INSTALL`, then touch `Start Applic`.

Step 12. After the Install program has been loaded, remove the disc from drive A and insert your BASIC master into drive A.

Step 13. Touch `Install Applics`.

Step 14. Touch `Show Applics`.

Step 15. Select `BASIC` by touching this field.

Step 16. Touch `Start Install`.

Step 17. After the installation procedure finishes, touch `Exit Select`.

NOTE

Step 18 applies to the initial version of P.A.M. (version number A.01.02). If you have a later version of P.A.M., proceed to Step 19.

Step 18. Touch `Exit Install`. This is your last step when using version number A.01.02 of P.A.M..

Step 19. Touch `Main Menu`, and after the Main Menu appears touch `Exit Main`.

You have now successfully installed P.A.M. and BASIC on a single back-up disc.

For Hard Disc Drive Users

This section details the steps that you must take to place a working copy of Series 100/BASIC on a hard disc.

For this procedure, you need the following discs:

- Your back-up copy of the HP 150 SYS__MASTER
- Your back-up copy of DISC APPLICATIONS
- Your master copy of Series 100/BASIC
- Your hard disc drive

Step 1. If you have not already done so, format your hard disc. (The owner's manual for your hard disc supplies the necessary details.)

Step 2. Put your back-up copy of the SYS__MASTER (the disc containing P.A.M.) into the flexible disc drive A and bring up the main P.A.M. menu.

Step 3. Remove the SYS__MASTER disc and insert your copy of DISC APPLICATIONS into the flexible disc drive A. (This disc contains the "Install" utility.)

Step 4. Touch `Reread Discs`.

P.A.M. updates the list of possible selections to include the INSTALL utility.

Step 5. Touch the `INSTALL` field to select this option. (You know you have successfully selected Install when the field becomes highlighted.)

Step 6. Touch `Start Applic`.

The message `Loading Install` appears on your screen.

Step 7. When the red light on the disc drive goes out, remove the DISC APPLICATIONS disc and insert your Series 100/BASIC master disc.

Step 8. Inspect the current screen display. The message `Select a function below` appears on the screen. Since you want to install an application program and not remove one, touch `Install Applics`.

Step 9. The next screen asks you to select the correct disc drives. Remember, you are installing BASIC "FROM:" the flexible disc drive A "TO:" the fixed disc drive B.

Step 10. Touch "`A`" in the FROM column until this field is highlighted.

Step 11. Touch "`B`" in the TO column until this field is highlighted.

Step 12. Touch `Show Applics`.

The message `Select the applications to be installed` appears on the screen.

Step 13. Select `BASIC` by touching this field.

Step 14. Touch `Start Install`.

Step 15. When the installation procedure finishes, the message `Install completed` appears on the screen. Touch `Exit Select`.

NOTE

Step 16 applies to the initial version of P.A.M. (version number A.01.02). If you have a later version of P.A.M., proceed to Step 17.

Step 16. Touch `Exit Install`. This is your last step when using version number A.01.02 of P.A.M..

Step 17. Touch `Main Menu`, and after the Main Menu appears touch `Exit Main`.

Starting BASIC

After you have both the operating system and BASIC on a single disc, running BASIC becomes a simple task. You only need to insert this disc into drive A, simultaneously press the [Shift], [CTRL], and [Reset] keys to "reboot" the system, and touch Start Applic to load BASIC into your computer's memory. (Refer to Chapter 3 for information on increasing your flexibility when entering BASIC.)

Index

A

ABS 6-3
Absolute Value 6-3
Adding Text 1-8, 1-11
Algebraic Expressions 2-11
Alphabetizing Strings 2-17
Altering Data And Variables 5-6
AND 2-13
Arctangent 6-3
Arithmetic Functions 5-14
Arithmetic Functions, Derived 5-15
Arithmetic Operators 2-10
Arithmetic Overflow 2-12
Array Variables 2-6
Arrays, Deleting 6-42
Arrays, Dimensioning 6-37
Arrays, Initial Subscript 6-104
ASC 6-3
Assembly Language Subroutines D-1
Assigning Values To Variables 6-77
Asterisk After Line Number 6-4
ATN 6-3
AUTO 6-4

B

Back-up Copy For BASIC 1-1, E-1, F-1
BASIC 3-2
BASIC Command Line 3-2
BASIC Functions 5-12
Bits, Masking 2-15
Bits, Merging 2-15
BLOAD 6-6
Braces xi
Brackets xi
Branching Statements 5-8
Branching To Another Program 5-9
Branching, Conditional 5-9
Branching, Unconditional 5-9
BSAVE 6-8

C

CALL 6-9, D-3
CALLS 6-10
Capital Letters xi
CDBL 6-11
CHAIN 6-12
Chapter Format 6-2
Character Comparisons 2-17
Character Set 1-5
CHR$ 6-17
CINT 6-17
CLEAR 6-18
Clearing The Screen 5-7
CLOSE 6-20

Colon As Statement Separator 1-5
Column Position 6-85
Command Level 1-2
Commands Used As Program Statements 5-4
COMMON 6-21
Computer Control Statements 5-7
Concatenation 2-17
Conditional Branching Statements 5-9
Constants 2-2
CONT 6-23
Control Characters 1-7, 6-95
Control-C, Cancelling AUTO 6-4
Control-C, Cancelling INKEY$ 6-65
Control-C, Cancelling LINE INPUT 6-78
Control-C, Cancelling LIST 6-81
Control-C, Cancelling WAIT 6-151
Control-C, Returning To Command Level 1-7, 6-142
COS 6-25
Cosecant 5-15
Cotangent 5-15
CSNG 6-25
CVD 6-26
CVI 6-26
CVS 6-26

D

DATA 6-27
Data Operators 2-1
DATA Statements, Rereading 6-27, 6-130
Data Variables 2-1
DATE$ Function 6-28
DATE$ Statement 6-29
Debugging Statements 5-11
Declaration Characters 2-4, 2-5
DEF FN 6-30
DEF SEG 6-32
DEF USR 6-33
DEFDBL 6-34
DEFINT 6-34
DEFSNG 6-34
DEFSTR 6-34
Defining Data Or Variables 5-6
Defining Error Codes 6-45
DELETE 6-36
Deleting Text 1-12
Derived Functions 5-15
DIM 6-37
Direct Mode 1-2
Disc File Names 4-1
Division By Zero 2-12
Documenting Your Program 1-18
Double Precision 2-3

E

e 6-47
EDIT 6-38
Edit Keys 1-7
Edit Mode Subcommands 1-9
Ellipsis xi
ELSE 6-61
END 6-39
Entering A Program 1-8
EOF 6-40
Equality Testing 6-63
EQV (Equivalent) 2-15
ERASE 6-42
Erasing Text 1-12
ERL 6-43
ERR 6-43
ERROR 6-45
Error Codes A-1
Error Codes, Defining 6-45
Error Messages 1-18, A-1
Escape Sequences B-1
Evaluation Order 2-10, 2-13
Exchanging Values 6-141
Exclusive OR 2-14
Executable Statement 1-5
EXP 6-47
Exponent, Floating Point 2-2
Exponential Format 6-115
Exponentiation 2-10, 6-47
Expressions 2-10

F

"Falling Through" 6-57
FIELD 6-48
File Operations 4-1, 5-5
FILES 6-50
Finding Text 1-12
FIX 6-51
Fixed Point Constants 2-2
Floating Point Constants 2-2
FOR 6-52
FOR/NEXT Loops 5-9, 6-52
Format For Functions 6-2
Format For Instructions 6-2
Formatting A Program Line 1-4
Formatting Numbers 6-113
Formatting Strings 6-112
Formatting The Random-file Buffer 6-48
FRE 6-55
Functional Operators 2-16

G

General Purpose Functions 5-13
Generating Line Numbers Automatically 6-4
GET 6-56
GOSUB 6-57
GOTO 6-59

H

Hex Constants 2-2
HEX$ 6-60
Hyperbolic Trigonmetric Functions 5-15

I

IF 6-61
IMP (Implied) 2-14
Inclusive OR 2-14
Indirect Mode 1-3
Infinite Loop With WAIT 6-151
INKEY$ 6-65
INP 6-65
INPUT 6-66
Input Statements 5-10
Input Editing 1-11
Input/Output Functions 5-13
INPUT# 6-69
INPUT$ 6-71
Inserting Text 1-11
Installing BASIC 1-1, E-1, F-1
INSTR 6-72
Instructions x
INT 6-73
Integer Constants 2-2
Integer Division 2-11
Inverse Trigonmetric Functions 5-15
Italicized Words xi

J

Justifying Text 6-87

K

KILL 6-74

L

LEFT$ 6-76
Left-justifying A String 6-87
LEN 6-76
LET 6-77
Line Format 1-4
LINE INPUT 6-78
LINE INPUT# 6-79
Line Modify 1-15
LIST 6-81
LLIST 6-81
LOAD 6-83
LOC 6-84
LOF 6-84
LOG 6-85
Logical Line 1-4
Logical Operators 2-13
Looping Statements 5-9
Lower Case Letters xi
LPOS 6-85
LPRINT 6-86
LPRINT USING 6-86
LSET 6-118

M

Machine Infinity 2-12
Making A Backup Copy For BASIC 1-1
Mantissa, Floating Point 2-2
Masking Bits 2-15
Memory Allocation D-2
Memory Image File 6-6
MERGE 6-88
Merging Bits 2-15
MID$ Function 6-90
MID$ Statement 6-91
MKD$ 6-92
MKI$ 6-92
MKS$ 6-92
MOD 2-12
Modes Of Operation 1-2
Modify Mode 1-14
Modifying Text 1-9
Modulus Arithmetic 2-11
Moving The Cursor 1-10

N

NAME 6-93
Natural Logarithms 6-47, 6-85
Nesting FOR Loops 6-52
Nesting IF Statements 6-62
Nesting Subroutines 6-57
Nesting WHILE Loops 6-152
NEW 6-94
NEXT 6-52
Non-executable Statement 1-5
NOT 2-13
Notation Conventions xi
NULL 6-95
Numeric Fields 6-113
Numeric Variables 2-5

O

Octal Constants 2-3
OCT$ 6-96
ON ERROR GOTO 6-97
ON...GOSUB 6-99
ON...GOTO 6-100
OPEN 6-101
Operators 2-10
OPTION BASE 6-104
OR 2-14
Order Of Precedence 2-10, 2-13
OUT 6-105
Output Statements 5-10
Output Functions 5-13
Overflow In Arithmetic Operations 2-12

P

Parentheses And Order Of Evaluation 2-11
PEEK 6-106
POKE 6-107
POS 6-108
Preface x
PRINT 6-109
Print Operations 1-19
PRINT USING 6-112
Print Zones 6-109
PRINT# 6-117
PRINT# USING 6-117
Printing Numbers 6-113
Printing Strings 6-112
Program Control Statements 5-8
Program Lines 1-4
Programming Guidelines 1-1
Programming Tasks 5-1
Protected Files 4-8
Punctuation xi
PUT 6-120

Q

Question Mark 1-3, 6-109
Question Mark Prompt 6-66, 6-78
Question Mark Prompt, Suppressing 6-66, 6-69
Quick Computation 1-3

R

"Random" Numbers 6-133
Random-Access Files 4-3
RANDOMIZE 6-121
READ 6-123
Reference Tables C-1
Relational Operators 2-12
REM 6-125
RENUM 6-127
Replacing Text 1-12
Rereading DATA Statements 6-27, 6-130
Reseeding Random-number Generator 6-121
Reserved Words 2-4, C-10
RESET 6-129
RESTORE 6-130
RESTORE With CHAIN 6-14
RESUME 6-131
RETURN 6-132
RIGHT$ 6-133
Right-justifying A String 6-87
RND 6-133
RSET 6-87
RUN 6-134

S

SAVE 6-135
Secant 5-15
Sequential Files 4-1
SGN 6-136
SIN 6-136
Single Precision 2-3
Space Bar 1-11
SPACE$ 6-137
SPC 6-137
Special Functions 5-17
Square Brackets xi
SQR 6-138
Start of Text Pointer 1-16
Starting BASIC 1-2
STEP With FOR Statement 6-52
STOP 6-139
STR$ 6-140
String Fields 6-112
String Functions 5-16
String Operations 2-17
String Operators 2-17
String Variables 2-5
STRING$ 6-140
Subroutine Statements 5-9
SWAP 6-141
SYSTEM 6-142
System Commands 5-3

T

TAB 6-143
TAN 6-143
Terminal I/O Statements 5-10
Testing Equality 6-63
THEN 6-61
TIME$ Function 6-144
TIME$ Statement 6-145
Trace Flag 6-146
TROFF 6-146
TRON 6-146
Truth Tables 2-13
Two's Complement 2-16
Type Conversion 2-7
Type Declaration Characters 2-4, 6-34

U

User-defined Functions 6-33
Using Commands As Program Statements 5-4
USR Function 6-147, D-8

V

VAL Function 6-148
Variable Length String Field 6-113
Variables 2-3
VARPTR 6-149
Vertical Bar (|) xi

W

WAIT 6-151
WHILE...WEND 6-152
WIDTH 6-154
Wild Cards 6-50, 6-74
WRITE 6-155
WRITE# 6-156
Writing A Simple Program 1-20

X

XOR 2-14