Model 2000

# BASIC

TRS-80

# Running GRAPHICS.BAS:

## A Demonstration of High Resolution Graphics

We have added a file, GRAPHICS.BAS, to your MS-DOS system diskette. This program demonstrates some of the high resolution graphics available with BASIC. To use the program, you need the CM-1 Color Monitor and a Monochrome Graphics Option board upgraded with the Color Graphics Option kit.

**Note:** If you have a VM-1 Monochrome Monitor and the Monochrome Graphics Option Board, you can run the program after removing the color-related statements and changing the SCREEN 3 statements to SCREEN 4. (See *Model 2000 BASIC Reference* for more information on these statements.)

If you do not plan to use GRAPHICS.BAS, you may delete the file from the diskette, as described in Chapter 8 of *Introduction to the Model 2000*. Otherwise, run the file by typing (at the system prompt):

```
BASIC GRAPHICS.BAS (ENTER)
```

When finished, return to MS-DOS by typing:

```
SYSTEM (ENTER)
```

Tandy Corporation
Part No. 8759267

Cat. No. 26-5103

# BASIC

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK AND TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

# LIMITED WARRANTY

## I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

## II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Radio Shack and Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores and from Radio Shack franchisees and dealers at its authorized location.** The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **Except as provided herein, Radio Shack makes no express warranties, and any implied warranty of merchantability or fitness for a particular purpose is limited in its duration to the duration of the written limited warranties set forth herein.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

## III. LIMITATION OF LIABILITY

A. **Except as provided herein, Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by "Equipment" or "Software" sold, leased, licensed or furnished by Radio Shack, including, but not limited to, any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of the "Equipment" or "Software". In no event shall Radio Shack be liable for loss of profits, or any indirect, special, or consequential damages arising out of any breach of this warranty or in any manner arising out of or connected with the sale, lease, license, use or anticipated use of the "Equipment" or "Software".**

**Notwithstanding the above limitations and warranties, Radio Shack's liability hereunder for damages incurred by customer or others shall not exceed the amount paid by customer for the particular "Equipment" or "Software" involved.**

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

## IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

## V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

## VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

GW™-BASIC Software: Copyright 1983 Microsoft Corporation.
Licensed to Tandy Corporation. All Rights Reserved.

MS™-DOS Software: Copyright 1983 Microsoft Corporation.
Licensed to Tandy Corporation. All Rights Reserved.

Model 2000 BIOS Software: Copyright 1983 Tandy Corporation.
All Rights Reserved.

BASIC Reference Manual: Copyright 1983 Microsoft Corporation and
Tandy Corporation.
All Rights Reserved.

Reproduction or use without express written permission from Tandy Corporation, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information contained herein.

10 9 8 7 6 5 4 3

# Introduction

This manual is about the popular GW-BASIC from Microsoft. BASIC for MS-DOS is an "interpreter." When you run a program, it executes each statement one at a time. This makes it quick and easy to use. It also allows you to take advantage of many MS-DOS features, such as:

- Faster running programs
- Expanded graphics capabilities

# About this Manual

This is a reference manual, not a tutorial. We assume you already know BASIC and are using this manual to quickly find the information you need. If you do not know BASIC, many excellent books are available at your local bookstore written in a tutorial fashion to teach you BASIC.

> Section I — Operations. This section shows how to load BASIC. It also demonstrates how to write, run, and save a BASIC program on disk.
>
> Section II — The BASIC Language. This section includes a definition for each BASIC keyword (statements and functions) in alphabetical order. In addition, it shows how to write a program to store data on disk.

## Notations

| | |
|---|---|
| CAPITALS | material that must be entered exactly as it appears. |
| *italics* | words, letters, characters, or values you must supply from a set of acceptable entries. |
| . . . (ellipsis) | items preceding the ellipsis may be repeated. |
| X'NNNN | NNNN is a hexadecimal number. |
| O'NNNN | NNNN is an octal number. |

| | |
|---|---|
| **(KEYNAME)** | a key on your keyboard. |
| ƀ | a blank space character (ASCII code 32). For example, in<br><br>BASICƀƀPROG<br><br>two spaces are between BASIC and PROG. |

# Terms

| | |
|---|---|
| *buffer* | a number in the range 1 to 15. This refers to an area in memory that BASIC uses to create and access a disk file. Once you use a buffer to create a file, you cannot use it to create or access any other files; you must first close the file. You may only access an open file with the buffer used to open it. |
| [*parameters*] | information you supply to specify how a command is to operate. Parameters enclosed in brackets are optional. |
| [*arguments*] | expressions you supply for a function to evaluate. Arguments enclosed in brackets are optional. |
| syntax | a command with its parameter(s), or a function with its argument(s). This shows the format to use for entering a keyword in a program line. |
| *line* | a numeric expression that identifies a BASIC program line. Each line has a number between 0 and 65529. |
| *integer* | any integer expression. It may consist of an integer or of several integers joined by operators. Integers are whole numbers between −32768 and 32767. |

| | |
|---|---|
| *string* | any string expression. It may consist of a string, or of several strings joined by operators. A string is a sequence of characters that is to be taken verbatim. |
| *number* | any numeric expression. It may consist of a number or of several numbers joined by operators. |
| *dummy number* or *dummy string* | a number (or string) used in an expression to meet syntactic requirements, but the value of which is insignificant. |

# Table of Contents


## Section I / Operations . . . 7

### Chapter 1 / Sample Session . . . 8

Loading BASIC . . . 8
Options for Loading BASIC . . . 10
Typing the Program . . . 11
Saving the Program . . . 12
Loading the Program . . . 14

### Chapter 2 / Command and Execution Modes . . . 15

Command Mode . . . 15
  Interpretation of a Line . . . 15
  Immediate Lines . . . 16
  Program Lines . . . 16
  Special Keys in Command Mode . . . 17
Execution Mode . . . 19
  Special Keys in Execution Mode . . . 19

### Chapter 3 / The Line Editor . . . 21

Special Keys in the Edit Mode . . . 22
Changing Lines Anywhere on the Screen . . . 26
More on Line Edit Mode . . . 27

## Section II / The BASIC Language . . . 28

### Chapter 4 / BASIC Concepts . . . 31

Overview: Elements of a Program . . . 31
How BASIC Handles Data . . . 34
  How BASIC Classifies Constants . . . 43
  How BASIC Classifies Variables . . . 46
  How BASIC Converts Numeric Data . . . 47
How BASIC Manipulates Data . . . 51
  Operators . . . 51
  Functions . . . 61
How to Construct an Expression . . . 62

### Chapter 5 / Disk Files . . . 65

Sequential Access Files . . . 65
  Creating a Sequential File . . . 66
  Updating a Sequential File . . . 68
Direct Access Files . . . 70
  Creating a Direct File . . . 70
  Accessing a Direct File . . . 72

**Chapter 6 / Introduction to Keywords** . . . . . 75
Format For Chapter 7 . . . . . 75
Statements . . . . . 76
Functions . . . . . 81
Introduction to Graphics . . . . . 84
Graphics . . . . . 84
Medium Resolution Color Graphics Option . . . . . 85
High Resolution Monochrome Graphics Option . . . . . 85
High Resolution Color Graphics Option . . . . . 87
Specifying Coordinates . . . . . 88
Aspect Ratio . . . . . 89
Screen Mode 1 . . . . . 90
Screen Mode 2 . . . . . 90
Screen Modes 3 and 4 . . . . . 91
**Chapter 7 / BASIC Keywords** . . . . . 93
**Section III / Appendices** . . . . . 331
**Appendix A / Error Codes and Messages** . . . . . 333
**Appendix B / BASIC Reserved Words and Derived Functions** . . . . . 341
**Appendix C / Video Display Worksheet** . . . . . 344
**Appendix D / Memory Map** . . . . . 345
**Appendix E / Technical Functions** . . . . . 347
Interfacing with Assembly Language Subroutines . . . . . 347
Accessing String Variables . . . . . 354
File Control Block . . . . . 356
How Variables Are Stored . . . . . 358

# Using BASIC

# Section I

# Operations

Using BASIC

# Chapter 1

# Sample Session

The easiest way to learn how BASIC operates is to write and run a program. This chapter provides sample statements and instructions to help familiarize you with the way BASIC works.

The main steps in running a program are:

A) Loading BASIC
B) Typing the program
C) Editing the program
D) Running the program
E) Saving the program on disk
F) Loading the program back into memory

# Loading BASIC

We recommend that you read your *Introduction to Model 2000* for complete startup information on your Model 2000 before you load BASIC. It details the necessary steps to get to the MS-DOS command level prompt.

At the MS-DOS system prompt A>, you can load BASIC into the computer's memory by typing

BASIC (ENTER)

A paragraph with copyright information appears on your screen, followed by: Ok

You may now begin using BASIC.

# Options for Loading BASIC

When loading BASIC, you can also specify a set of options. They are:

BASIC [*filename*] [/F:*number of files*] [/C:*buffer size*] [/M:*highest memory location*] [/S:*record length*]

*Filename* specifies a program to run immediately after BASIC is started.

*/F:* specifies the maximum number of data files that may be open at any one time (from 0-15). If you omit this option, the number of files defaults to three.

Each file you specify may use up to 190 bytes of memory. Sequential access files always use 190 bytes of memory. The amount of memory a direct access file uses depends on the record size set with the /S: option. Each file uses 62 bytes of memory for the file control block, plus the record size. For example, if you specify a record size of 50 with the /S: switch, the file uses 112 bytes.

*/C:* specifies the size of the receive buffer for RS232 communication. If you omit the /C: option, BASIC allocates 256 bytes for the receive buffer. The transmit buffer is always 128 bytes.

*/M:* specifies the highest memory location for BASIC to use. Omit this option unless you plan to call assembly-language subroutines. (In that case, you may want to set the highest memory location well below the top of BASIC's data segment.) If you omit this option, the system allocates 64K bytes of memory to BASIC.

*/S:* specifies the maximum record size for direct access files. If you omit the /S: option, BASIC assumes 128 bytes.

## Examples

A>
BASIC PAYROLL /F:5 ENTER

initializes BASIC, then loads and runs the program PAYROLL; allows five data files to be open; uses all memory available.

A>
BASIC /M:21000 ENTER

initializes BASIC; allows three data files to be open; sets the highest memory location to be used by BASIC at 21000, the first 21K bytes of BASIC's data segment.

A>
BASIC /M:21000 /F:6 ENTER

initializes BASIC; sets the highest memory location at 21000; allows six data files to be open. Notice that the sequence in which the /M: and /F: options are specified is irrelevant.

A>
BASIC

initializes BASIC; allows three data files to be open; uses all memory available.

# Typing the Program

Let's write a small BASIC program. Before pressing ENTER after each line, check the spelling. If you have made any mistakes, use the ← key to correct them.

```
10 A$="WILLIAM SHAKESPEARE WROTE " ENTER
15 B$="THE MERCHANT OF VENICE" ENTER
20 PRINT A$; B$ ENTER
```

Check your spelling again. If it is still not perfect, enter the line number where you made the mistake. Then type the entire line again.

For example, suppose you had typed:

15 B$ = "THE VERCHANT OF VENICE"

To correct line 15, retype it:

15 B$ = "THE MERCHANT OF VENICE" (ENTER)

Then type:

RUN (ENTER)

Your screen should display:

WILLIAM SHAKESPEARE WROTE THE MERCHANT OF VENICE

BASIC replaced line 15 in the original program with the most recent line 15.

> **Note:** BASIC "reads" your program lines in numerical order. It doesn't matter if you entered line 15 after line 20; BASIC still reads and executes 15 before "looking" at 20.

BASIC has a powerful set of commands that allow you to correct mistakes without retyping the entire line. These commands are discussed in Chapter 3, the "Line Edit Mode."

# Saving the Program on Disk

You can save any BASIC program on disk. To do this, you assign it a *filespec*. The filespec tells BASIC on which disk you want to save the file and the name of the file. A filespec consists of drive identifier, filename, and extension. Only the filename is required. The filespec must be enclosed in double quotes.

Filenames must conform to the MS-DOS file naming conventions. A filename can have a maximum of eight alphanumeric characters. The first character must be a letter, A through Z. The remaining seven characters may be any of the following:

the letters A through Z
the digits 0 through 9
the special characters <, >, (, ), {, }, @, #, $, %, ^, &, !, -, _, ', and /

The extension may be up to three characters long and must also begin with a letter. The other two characters may be any of the characters that are allowed in the filename. A period (.) must be included between the filename and the extension. If you omit an extension with the SAVE, LOAD, MERGE, and RUN commands, BASIC appends the extension **.BAS**.

The drive identifier specifies on which disk you want BASIC to save your program. The drive identifier precedes the filename and extension and may be any of the valid drive letters (A through D); it must be followed by a colon. If you omit the drive identifier, BASIC saves the file on the MS-DOS current drive.

For example, to save the program we just wrote on Drive B, assign it the filename "AUTHOR.BAS". Type the following command:

SAVE "B:AUTHOR.BAS" ENTER

It takes a few seconds for the computer to find a place on disk to store a program and to copy the program from memory to the disk. When the program is saved on the disk, the screen displays Ok.

## Examples

SAVE "AUTHOR.WIL" ENTER

saves the program under the filename AUTHOR, with the extension .WIL on the MS-DOS current drive.

SAVE "A:AUTHOR" ENTER

saves the program under the filename "AUTHOR" on the disk in Drive A. Because no extension is given, BASIC appends the extension .BAS.

# Loading the Program

If, after writing or running other programs, you want to use this program again, you must "load" it back into memory. To do this, type:

LOAD "*filespec*", R

## Example

LOAD "AUTHOR", R (ENTER)

tells the computer to load the program "AUTHOR" from disk into memory; option R tells the computer to run it.

Another way to load and run a program is to type:

RUN "*filespec*"

RUN automatically loads and runs the program specified by "*filespec*".

The SAVE, LOAD, and RUN commands are discussed in more detail in Chapter 7.

# Chapter 2

# Command And Execution Modes

This chapter describes BASIC's command and execution modes. BASIC is in the command mode when you are typing in program lines and immediate lines. BASIC is in the execution mode when it is performing the instructions in the program and immediate lines.

# Command Mode

Whenever you enter the command mode, BASIC displays the prompt:

In the command mode, BASIC does not "read" your input until you complete a "logical line" by pressing (ENTER). This is called "line input," as opposed to "character input."

A logical line is a string of up to 255 characters and is always terminated by pressing (ENTER). Of these 255 characters, 249 are reserved for the line itself; the other six are reserved for the line number and the space following the line number.

A physical line, on the other hand, is one line on the display. It contains a maximum of 80 characters.

For example, if you type 100 R's and then press (ENTER), you have two physical lines, but only one logical line.

## Interpretation of a Line

BASIC always ignores leading spaces in the line — it jumps ahead to the first non space character. If this character is not a digit, BASIC treats the line as an immediate line. If it is a digit, BASIC treats the line as a program line.

For example, if you type:

```
PRINT "THE TIME IS" TIME$ (ENTER)
```

BASIC takes this as an immediate line.

But if you type:

10 PRINT "THE TIME IS" TIME$ (ENTER)

BASIC takes this as a program line.

## Immediate Lines

An immediate line consists of one or more statements separated by colons. There is no line number in an immediate line. The line is executed as soon as you press (ENTER). After BASIC executes the line, it is no longer in memory. The values of variables and constants are still in memory, but the statement no longer exists. Immediate lines are useful for using the computer as a calculator for quick computations that don't require an entire program. For example:

```
Ok
MILES = 133:GAS=11:MPG=MILES/GAS
```

After this statement is executed, the value of the variables MILES, GAS, and MPG still exist in memory. But the instruction itself does not.

```
Ok
CLS: PRINT "THE SQUARE ROOT OF 2 IS" SQR(2)
```

is an immediate line. When you press (ENTER), BASIC executes it.

## Program Lines

A program line consists of a line number in the range 0 to 65529, followed by one or more statements separated by colons. When you press (ENTER), the line is stored in memory. All lines that you enter with a line number are stored in memory until you execute a RUN or other execute command. For example:

```
100 CLS: PRINT "THE SQUARE ROOT OF 2 IS"
SQR(2)
```

is a program line. When you press (ENTER), BASIC stores it in memory. To execute it, type:

RUN (ENTER)

> **Note:** If you include numeric constants in a line, BASIC evaluates them as soon as you press (ENTER); it does not wait until you RUN the program. If any numbers are out of range for their type, BASIC returns an error message immediately after you press (ENTER).

## Special Keys

The CTRL and ALT keys have special functions in BASIC. When you press and hold one of these keys while typing another key, BASIC performs functions to make entering and editing program lines and immediate lines easier. These keys have no function in BASIC unless they are pressed with another key. For example, if you press (CTRL) key and type (H), BASIC backspaces and deletes the character.

## Special Keys in the Command Mode

| | |
|---|---|
| (BACKSPACE) or (CTRL)(H) | Backspaces the cursor, erasing the preceding character in the line. Use this to correct typing errors before pressing (ENTER). |
| (SPACE BAR) | Enters a blank space character and advances the cursor. |
| (BREAK) or (CTRL)(C) | Interrupts line entry and starts over with a new line. |
| (CTRL)(J) | Line feed — starts a new physical line without ending the current logical line. |
| (CAPS) | Switches the display to either all uppercase or uppercase/lowercase mode. |
| (ENTER) or (CTRL)(M) | Ends the current logical line. BASIC "takes" the line. |

| CTRL U or ESC | Erases the current line. |
|---|---|
| CTRL ] or ← | Moves the cursor one position to the left. |
| CTRL \ or → | Moves the cursor one position to the right. |
| CTRL F or CTRL → | Moves the cursor to the first character in the next word to the right of the current cursor positon. |
| CTRL B or CTRL ← | Moves the cursor to the first character in the word to the left of the current cursor position. |
| CTRL R or INSERT | Turns on insert mode if it is off; turns off insert mode if it is on. |
| DELETE | Deletes the character at the current cursor position. |
| CTRL W | Deletes the next word to the right of the cursor. |
| CTRL T | Displays the soft key values of the 12 Function Keys. |
| END or CTRL N | Moves the cursor to the last character in the logical line. |
| CTRL END or CTRL E | Deletes all characters from the current cursor position to the end of the line. |
| CTRL I or TAB | Advances the cursor to the next tab position. Tab positions are set at every eight character positions. |
| CTRL G | Rings the bell at the terminal. |

Some BASIC keywords are associated with alphabetic characters (A-Z). To enter these keywords easily, press (ALT) and the corresponding letter. BASIC inserts the keyword at the current cursor position. The keywords and their associated letters are listed below. Letters that don't have an associated keyword are indicated by "(none)."

| | | | |
|---|---|---|---|
| A | AUTO | N | NEXT |
| B | BSAVE | O | OPEN |
| C | COLOR | P | PRINT |
| D | DELETE | Q | (none) |
| E | ELSE | R | RUN |
| F | FOR | S | SCREEN |
| G | GOTO | T | THEN |
| H | HEX$ | U | USING |
| I | INPUT | V | VAL |
| J | (none) | W | WIDTH |
| K | KEY | X | XOR |
| L | LOCATE | Y | (none) |
| M | MOTOR* | Z | (none) |

*MOTOR is a reserved word, but is not a recognized statement in this implementation of BASIC.

# Execution Mode

When BASIC is executing statements (immediate lines or programs), it is in the execution mode. In this mode, the contents of the video display are under program control.

## Special Keys in the Execution Mode

| | |
|---|---|
| (HOLD) or (CTRL)(S) | Pauses execution. Press any other key (except (BREAK)) to continue. |
| (BREAK) | Terminates execution and returns you to command mode. |
| (ENTER) or (CTRL)(M) | Interprets data entered from the keyboard as a response to the INPUT statement. |

# Chapter 3

# The Line Editor

Your BASIC Editor lets you "debug" (correct errors in) your BASIC program quickly and efficiently without retyping entire lines.

To enter line edit mode type :

EDIT *line number* (ENTER)

This lets you edit the specified line number. (If the line number you specify has not been used, an "Undefined line number" error occurs.)

You may also use the LIST command to list one or several lines before you make the changes. If you LIST one line you can use the keys described in the next section to make changes to the line. If you LIST several lines, see the last section of this chapter, "Changing Lines Anywhere on the Screen."

You may also type:

EDIT . (ENTER)

The period after EDIT means that you want to edit the current program line, the last line entered, the last line altered, or a line in which an error has occurred. Notice that you need to type a blank before the period; otherwise, BASIC gives you a "Syntax error" message.

For example, type the following line and press (ENTER).

100 PRINT "This is our example line."

This line will be used in exercising all the edit subcommands described below.

Now type EDIT 100 and press (ENTER). BASIC displays the entire line and positions the cursor under the first digit of the line number. This starts the editor. You may now begin editing line 100. Line 100 can be modified by using any of the special keys described below. **Note:** None of the changes you make to a program line are entered until you press (ENTER).

# Special Keys in the Edit Mode

| Key | Function |
|---|---|
| (ENTER) or (CTRL)(M) | Records the changes you made in the current line and returns you to the command mode. |
| (←) or (CTRL)(]) | Moves the cursor one position to the left. If you advance the cursor past the left-hand margin of the screen, it moves to the right-hand margin of the screen on the previous line. |
| (→) or (CTRL)(\) | Moves the cursor one position to the right. If you advance the cursor past the right-hand margin of the screen, it moves to the left-hand margin of the screen on the next line. |
| (SPACEBAR) | Changes the character to a blank and advances the cursor one position to the right. |
| (CTRL)(→) or (CTRL)(F) | Moves the cursor right to the next word. The next word is the next letter or number that follows a blank or a special character. |
| (CTRL)(←) or (CTRL)(B) | Moves the cursor to the first character in the previous word. The previous word is the next letter or number to the left of the cursor that precedes a blank or special character. |
| (END) or (CTRL)(N) | Moves the cursor to the last character in the logical line. |

| Key | Function |
| --- | --- |
| DELETE | Erases the character at the current cursor position. All characters to the right of the cursor and subsequent characters and lines within the logical line move left or up one position. |
| BACKSPACE or CTRL H | Erases the character to the left of the cursor. All characters to the right and subsequent characters and lines within the logical line move left or up one position. |
| CTRL END or CTRL E | Erases all characters from the current cursor position to the end of the logical line. |
| CTRL U or ESC | Erases the entire logical line from the screen. BASIC does not record in memory any of the changes made to the line. You may press either of these anywhere in the line to cancel changes made. |
| TAB or CTRL I | Moves the cursor to the next tab position. Tab positions are set at every eight positions. As the cursor advances to the next tab position, the characters in the positions it is tabbing over are printed. |
| BREAK or CTRL C | Returns to direct mode and does not record in memory any of the changes to the line currently being edited. |
| CTRL K or HOME | Moves the cursor to the first position in row one. |
| CTRL L | Clears the screen and positions the cursor at the first position in row one. |

| | |
|---|---|
| (INSERT) | Allows you to enter characters between other characters that are already in the line. To insert characters press (INSERT) and type the characters you want to insert. All other characters on the logical line move to the right or down each time you insert a character. If while you are inserting characters you press the (TAB) key, blanks are inserted from the current cursor position to the next tab position. After you insert all the characters, press (INSERT) again and continue editing the line. |
| (CTRL)(Z) | Clears the screen from the current cursor position to the end of the screen. |

## Sample Session

Type

**EDIT 100**

Use the right arrow to space across the line to the "T" in "This." Type lowercase "t" and then (ENTER).

Type

**LIST 100**

to see that BASIC stored your change in memory.

Use the edit command to edit the line again. Press (END) to position the cursor on the second set of double quotes. Press (INSERT) and type

**We inserted the second sentence.** (ENTER)

Use the list command to see the new statement that is stored in memory.

Type

EDIT . (ENTER)

to edit the line again. You may use the period (.) instead of the line number to edit the current line. Use (TAB) and (→) to position the cursor on the "i" in inserted. Press (CTRL)(END). BASIC deletes all the characters you inserted except "we" and the blank. Use (BACKSPACE) to delete the blank. Use (CTRL)(←) to position the cursor on the previous word. Press (DELETE) twice to delete "we." Press (INSERT), then (") to put the double quote in at the end of the statement. Use the list command to see the new line.

Now let's add another line to the program. Type

200 GOTO 100 (ENTER)
RUN

BASIC is in a loop printing your message repeatedly on the screen. Press (BREAK) to stop program execution.

Type

DELETE 200 (ENTER)

Line 200 is erased from memory.

Use the edit command to edit the line again. Use (→) to position the cursor on the "P" in PRINT. Press (SPACEBAR) to change the "P" to a blank. Press (CTRL)(→) to position the cursor on the "t" in "this." Press (T) to change the lowercase "t" to a capital "T." Instead of pressing (ENTER) after you make the changes, press (ESC). Use the list command to see that BASIC did not record your change, because you pressed (ESC) instead of (ENTER).

Now you have used all the special keys in line edit mode. If you still don't feel comfortable with them, go through the sample session again. If you feel confident that you understand the line editor, read on to learn about some special keys that make it easier and faster to change lines anywhere on the screen.

# Changing Lines Anywhere on the Screen

When more than one line is displayed on the screen, you may use the arrow keys to move the cursor around the screen to different program lines to correct errors. After you make all the corrections, you must go to the beginning of each line that you modified and press (ENTER) to record the change in memory. Using the arrow keys to make corrections can be much quicker than typing EDIT and a line number for every line that needs to be changed.

| | |
|---|---|
| (CTRL)(^) or (↑) | Moves the cursor up one row to the character above the current cursor position. |
| (CTRL)(–) or (↓) | Moves the cursor down one row to the character below the current cursor position. |

You may also use the left and right arrow keys as previously described under "Special Keys in the Edit Mode."

## Sample Session

After you type each of the following lines, press (ENTER)

```
10 PRINT "With the fising cost of fuel, ras mileage ";
20 PRINT "has vecome an important donsideration"
30 PRINT "in the purchase of anew vehiclee"
```

Now you can use the arrow keys to correct the mistakes in the program statements.

1. Use (↑), then (→) to position the cursor on the "f" in "fising" in Line 10. Type "r" over the "f."
2. Use (TAB), then (→) to position the cursor on the "r" in "ras." Type (g) over the "r."
3. Use (↓), then (←) to position the cursor over the "d" in "donsideration." Type (c) over the "d."
4. Use (CTRL)(←), then (←) to position the cursor over the "v" in "vecome." Type (b) over the "v."

5. Use (↑), (CTRL)(→), and (→) to position the cursor over the "n" in "anew". Press (INSERT) and (SPACEBAR) to insert a blank between "a" and "new."
6. Use (→) to space across to the last "e" in "vehiclee." Press (DELETE) to erase the extra "e".
7. Use (←) to position the cursor over the "3" in Line 30. Press (ENTER).
8. Use (↑) to position the cursor over the "2" in Line 20. Press (ENTER).
9. Use (↑) to position the cursor over the "1" in Line 10. Press (ENTER).

# More on Line Edit Mode

If your computer encounters a syntax error while executing a program, BASIC automatically enters EDIT. It displays the line that contains the error. For example, type

**10 A = 2$12 (ENTER)**
**RUN**

The screen displays:

```
?Syntax error in 10
10 A = 2$12
```

EDIT positions the cursor under the first digit of the line number. Now press (→) to move the cursor to the dollar sign ($) and press (DELETE) and then (ENTER). BASIC stores the corrected line in memory.

BASIC Language

# Section II

# The BASIC Language

# Chapter 4

# BASIC Concepts

This chapter explains how to use the full power of BASIC for MS-DOS. This information can help programmers build powerful and efficient programs. If you are still something of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.

The chapter is divided into four sections:

**Overview — Elements of a Program.** This section defines many of the terms we use in the chapter.

**How BASIC Handles Data.** Here we discuss how BASIC classifies and stores data. This shows you how to get BASIC to store your data in its most efficient format.

**How BASIC Manipulates Data.** This gives you an overview of the operators and functions you can use to manipulate and test your data.

**How to Construct an Expression.** This topic can help you construct powerful statements that you can use instead of many short ones.

# Overview: Elements of a Program

This overview defines the elements of a program. A program is made up of "statements"; statements may have several "expressions."

## Program

A program is a group of instructions that perform a certain process. It is made up of one or more numbered lines. Each line contains one or more BASIC statements, the instructions. BASIC allows line numbers from 0 to 65529 inclusive. A line contains up to 255 characters, including the line number.* You may also have two or more statements to a line, separated by colons.

* You can type a maximum of 249 characters per line. BASIC reserves the remaining six characters for the line number and for the space following the line number.

Here is a sample program:

Line number → BASIC statement → Colon between statements → BASIC statement

```
100 CLS: LOCATE 2,30: PRINT "Graphic Characters"
110 FOR X = 1 to 6: PRINT CHR$(X),
120 NEXT X
130 FOR Y = 14 to 27: PRINT CHR$(Y),
140 NEXT Y
150 END
```

When BASIC executes a program, it handles statements one at a time, starting with the first and proceeding to the last. Some statements, such as GOTO, ON . . . GOTO, GOSUB, change this sequence.

## Statements

A statement is a complex instruction to BASIC, telling the computer to perform specific operations. For example:

```
GOTO 100
```

tells the computer to perform the operations of (1) locating line 100, (2) transferring control to that line and (3) executing the statement(s) on that line.

```
END
```

tells the computer to perform the operation of ending execution of the program.

Many statements instruct the computer to perform operations with data. For example, in the statement:

```
PRINT "SEPTEMBER REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the computer to print (display on the screen) the data inside quotes.

# Expressions

An expression is part of a statement. It is one or more pieces of data that produce a single value. The data may be an expression itself, or several pieces of data may be connected by operators or punctuation to form an expression. There are four types of expressions:

1. Numeric expressions, which contain numeric data.

Examples:

```
(1 + 5.2)/3
D
5*B
3.7682
ABS(X) + RND(0)
SIN(3 + E)
```

2. String expressions, which contain character data.

Examples:

```
A$
"STRING"
"STRING" + "DATA"
MO$ + "DATA"
MID$(A$,2,5) + MID$("MAN",1,2)
M$ + A$ + B$
```

3. Relational expressions, which test the relationship between two expressions.

Examples:

```
A=1
A$>B$
```

4. Logical expressions, which test the logical relationship between two expressions.

Examples:

```
A$="YES" AND B$="NO"
C>5 OR M<B OR 0>-2
578 AND 452
```

## Functions

Functions are automatic subroutines. Most BASIC functions perform computations on data. Some serve a special purpose, such as controlling the video display or providing data on the status of the computer. You may use functions in the same manner that you use any data, that is, as part of a statement.

These are some of BASIC's functions:

```
INT
ABS
STRING$
```

For example, ABS returns the absolute value of a numeric expression. The following example shows how this function works:

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

# How BASIC Handles Data

BASIC offers several methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we discuss the different ways BASIC represents data in your program. BASIC recognizes all data as either string or numeric. String and numeric values are represented as either constants or variables.

## Strings

A string is a sequence of up to 255 characters, enclosed in double quotes. You may store ASCII characters, graphics, or non-ASCII symbols in a string. Strings require three bytes of storage plus the number of characters in the string. For example, the string "TABBY" requires three bytes of storage plus five additional bytes to store the characters, for a total of eight bytes.

# Numerics

Numerics are positive or negative numbers. The five types of numerics are integer, fixed point, floating point, hexadecimal, and octal.

Integers are whole numbers between −32768 and 32767 that do not contain decimal points.

Examples:

**1 3200 −2 500 −12345**

Fixed point numbers are positive or negative real numbers and may contain decimal points.

Examples:

**1.1234 −100.999 .99 −.5998**

Floating point numerics are positive or negative numbers represented in exponential form (similar to scientific notation). A floating point numeric consists of a mantissa, followed by the letter E or D, and an exponent. The mantissa may be an integer or fixed point number. The letter E or D refers to number's numeric precision and means "times ten to the power of." We discuss numeric precision in the next section. The exponent is always an integer. Floating point numerics must be in the range 10−38 to 10+38. For example, in the number:

**2359E6**

2359 is the mantissa and 6 is the exponent. This number could be read as "2359 to sixth power," that is, 2359000000.

In the number

**235.988E−7**

235.988 is the mantissa and −7 is the exponent. This number could be read as "235.988 to the negative seventh power," that is, .0000235988.

Hexadecimal numerics are one- to four-digit hexadecimal representations of decimal numbers. Hexadecimal numerics are always preceded by the prefix &H, indicating that the numeric is a hexadecimal number and not a decimal number. The hexadecimal numbers are 0-9 and A-F.

Examples:

```
&H76 and &H032F
```

are hexadecimal representations of the decimal numbers 118 and 815 respectively.

Octal numbers are one- to six-digit octal values preceded by the prefix &O or just &. Although only the & is required, we recommend that you use &O for clarity in your program. The octal numbers are 0-7.

Examples:

```
&O123 and &O000456
```

are octal representations of the decimal numbers 83 and 302 respectively.

# Numeric Precision

Regardless of the numeric type, BASIC stores all numbers as integer, single precision, or double precision. The characteristics of a number determine its numeric precision. Numeric precision determines the amount of memory BASIC uses and the speed at which BASIC can process the number.

In this section we describe the different types of numeric precision and how BASIC automatically stores the number. Later we show you how to override the automatic storage by using type declaration tags.

## Integer

BASIC stores a number as an integer if it is in the range −32768 to +32767 and does not contain a decimal point. If it is outside the range, BASIC stores it as single or double precision in exponential format.

Integers require two bytes of storage. Integers require the least amount of storage space and are therefore faster for BASIC to access. But BASIC stores integers with the least degree of exactness.

For example:

```
1    3200    -2    500    -12345
```

can all be stored as integers.

## Single Precision

A single-precision number can include up to seven digits and may be in exponential form using E. If a number is larger than seven digits, BASIC stores it in double-precision form with D. Note that a single precision number may be either a fixed or a floating point numeric.

Single-precision numbers require four bytes of memory for storage. Even though BASIC stores the number with seven digits of precision, BASIC rounds the number to six digits when it is printed.

For example:

```
10.001    -200034    1.774E6    6.024E-23
123.4567
```

can all be stored as single-precision values.

## Double Precision

Double-precision numbers can include up to 16 digits and may be in the exponential form using D. Note that a double-precision number may be either a fixed or a floating point numeric.

Double-precision numbers require eight bytes of memory for storage. As with single precision, BASIC rounds the number to 16 digits when it is printed. Double-precision numbers require the most number of bytes and are therefore the slowest for BASIC to access. However, double-presicion numbers are the most exact.

For example:

```
1010234578
-8.7777651010
3.141592653589793
8.00100708D12
```

can all be stored as double-precision values.

## Constants

Constants are values input to a program that are not subject to change. The two types of constants are string and numeric. String constants must be enclosed in double quotes. Numeric constants can be integer, fixed point, floating point, hexadecimal, or octal.

The statement:

```
PRINT "NAME", "ADDRESS", "CITY", "STATE"
```

contains four string constants; NAME, ADDRESS, CITY, and STATE. Every time BASIC executes this PRINT statement, these four values are printed.

A numeric value that won't change in your BASIC program may be represented as either a string constant or a numeric constant. If you use puctuation in the number, it must be a string constant enclosed in double quotes. For example, in the statement:

```
PRINT "$250,000"
```

"$250,000" is a string constant.

In this statement:

```
PRINT "1,000 PLUS "; 1000; "EQUALS "; 2000
```

the first 1,000 is a string constant containing a comma. The other 1000 is a numeric constant.

# Variables

A variable is a place in memory where BASIC stores values that can change. This allows you to write programs that contain changing data.

In the statement:

```
A$ = "OCCUPATION"
```

The string variable A$ now contains the data OCCUPATION. However, if this statement appeared later in the program:

```
A$ = "FINANCE"
```

The variable A$ no longer contains OCCUPATION. It now contains the data FINANCE.

Strings with length zero are called "null" or "empty." Strings are useful for storing non numeric information such as names, addresses, or text.

Variables can also store numeric values. For example:

```
A = 134
```

The numeric variable A now contains the value 134. If this statement appears later in the program:

```
A = 100
```

the variable A now contains 100.

## Variable Names

In BASIC, variables are represented by names. Variable names can be up to 40 characters long, and they must begin with a letter, A through Z. This letter may be followed by any of the digits 0 through 9, a period, or a type declaration tag. Variable names cannot be exactly the same as any of the reserved words listed in Appendix B. However, reserved words may be imbedded in a variable name.

For example:

OR LEN OPTION

cannot be used as variable names. However,

**AM A A1 BALANCE**
**EMPLOYEE2 LEN2 OPTION1**

are all valid and distinct variable names.

Type Declaration Tags

A type declaration tag is a symbol at the end of a variable name that tells BASIC what kind of data the variable will store. The four types of declaration tags for variables are:

% Integer
! Single Precision
# Double Precision
$ String

For example:

INT% indicates to BASIC that the variable INT% will store integer type numerics.

PER! indicates to BASIC that the variable PER! will store single precision type numerics.

SPEED# indicates to BASIC that the variable SPEED# will store double precision type numerics.

NAME$ indicates to BASIC that the variable NAME$ will store string data.

## Arrays

An array is a group of related data values stored consecutively in memory. The entire group of data values are referred to as one variable name. Each value is called an *element* of the array. A *subscript* is an integer used to specify each element of the array. For example, an array named A may contain three elements referred to as:

A(Ø) A(1) A(2)

You may use each of these elements to store a separate data item, such as:

A(0) = .10
A(1) = .20
A(2) = .30

An array is similar to a table, such as a tax table. For example, array A could be the tax rate at different income levels. The tax rates are arranged in a row corresponding to the appropriate income levels, like this:

| Income | Tax Rate |
|---|---|
| 0 - 10,000 | .10 |
| 10,001 - 20,000 | .20 |
| 20,001 - 30,000 | .30 |

This is called a one-dimensional table because the elements are arranged in rows, one-dimension, with each dimension containing only one element.

Since tax rates are decreased by number of dependents, a tax array also has to contain columns within each row. As with a tax table, you can first locate the proper row, by income, and then move horizontally across the table to the appropriate column for number of dependents. An array that contains columns of data within rows of data is called a two-dimensional array. Each dimension of the two dimensions (row and column) contains more than one element.

A two-dimensional tax array named X could contain these elements:

| | | |
|---|---|---|
| X(0,0) = .15 | X(0,1) = .10 | X(0,2) = .05 |
| X(1,0) = .30 | X(1,1) = .25 | X(1,2) = .20 |
| X(2,0) = .45 | X(2,1) = .40 | X(2,2) = .35 |

The first subscript indicates the row number of the data and the second subscript indicates the column number. For example, the data stored in the second row at the second column, X(1,1) is .25. In a tax table these values are arranged like this:

| Income | Number of Dependents | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| 0 - 10,000 | .15 | .10 | .05 |
| 10,001 - 20,000 | .30 | .25 | .20 |
| 20,001 - 30,000 | .45 | .40 | .35 |

A taxpayer who has an income of $15,000 and two dependents has a tax rate of .25. That is element X(1,1) in the array.

If you further subdivide dependents column by the taxpayer's marital status, you need one more dimension within the column; depth. This is called a three-dimensional array because there are three dimensions (row, column, and depth), with each dimension containing more than one element. A three-dimensional array Z could contain these eight elements:

```
Z(0,0,0) = .05          Z(0,1,0) = .15
Z(0,0,1) = .10          Z(0,1,1) = .20

Z(1,0,0) = .25          Z(1,1,0) = .35
Z(1,0,1) = .30          Z(1,1,1) = .45
```

The first subscript indicates the row, the second subscript indicates the column, and the third subscript indicates the depth. In a tax table, these values are arranged like this:

| Income | Number of Dependents | | | |
|---|---|---|---|---|
| | 1 | | 2 | |
| | married | single | married | single |
| $0-10,000 | .05 | .10 | .15 | .20 |
| $10,001-$20,000 | .25 | .30 | .35 | .45 |

A taxpayer who has an income of $15,000, has one dependent, and is married has a tax rate of .30. That is element Z(1,0,1).

With BASIC, you may have up to 255 dimensions in your array and up to 32,767 elements in each dimension. Arrays may be of any type: string, integer, single-precision, or double-precision.

You may define arrays in your BASIC program with a DIM statement at the beginning of your program or just by setting the value of an element in the program. For example:

```
A(5) = 300
```

creates an array named A containing six elements and assigns element A(5) the value 300.

Use a DIM statement, to reserve space in memory for each element of the array. For example:

```
DIM C#(99)
```

creates array C and reserves memory for 100 single precision elements.

See the DIM statement in Chapter 6 for more information on creating arrays.

## How BASIC Classifies Constants

When BASIC encounters a data constant in a statement, it must determine the type of the constant: string, integer, single precision, or double precision. First, we list the rules BASIC uses to classify the constant. Then we show you how you can override these rules if you want to store a constant differently.

Rule 1

If the value is enclosed in double quotes, it is a string.

Examples:

```
"YES"
"3331 Waverly Way"
"1234567890"
```

Rule 2

If the value is not in quotes, it is a number.

Examples:

```
123001
1
-7.3214E + 6
```

Rule 3

Whole numbers in the range of −32768 to 32767 are integers.

Examples:

```
12350
-12
10012
```

**Note:** If you enter a number as a constant in response to a command that calls for an integer, and the number is out of integer range, BASIC converts the number to single or double precision. When the number is printed, it appears with a type-declaration tag at the end.

Rule 4

If the number is not an integer and contains seven or fewer digits, it is single precision.

Examples:

```
1234567
-1.23
1.3321
```

Rule 5

If the number contains more than seven digits, it is double precision.

Examples:

```
1234567890123456
−1000000000000.1
2.777000321
```

You can override BASIC's normal typing criteria by adding type declaration "tags" at the end of the numeric constant.

! Makes the number single precision. For example, in the statement:

```
A = 12.345678901234!
```

BASIC classifies the constant as single precision and shortens it to seven digits.

```
12.3457
```

E Single-precision exponential format. The E indicates that the constant is to be multiplied by a specific power of 10. For example:

```
A = 1.2E5
```

stores the single-precision number 120000 in A.

# Makes the number double precision. For example, in statement:

```
PRINT 3#/7
```

BASIC classifies the first constant as double precision before the division takes place.

D Double-precision exponential format. The D indicates the constant is to be multiplied by a specified power of 10. For example, in:

```
A = 1.23456789D−1
```

the double-precision constant has the value 0.123456789.

## How BASIC Classifies Variables

When BASIC encounters a variable name in the program, it classifies it as either a string, an integer, a single-precision number, or a double-precision number.

BASIC classifies all variable names as single-precision initially. For example:

```
AB          AMOUNT          XY          L
```

are all single precision initially. If this is the first line of your program:

```
LP = 1.2
```

BASIC classifies LP as a single-precision variable.

However, you may assign different attributes to variables by using definition statements at the beginning of your program:

DEFINT — Defines variables as integer
DEFDBL — Defines variables as double-precision
DEFSTR — Defines variables as string
DEFSNG — Defines variables as single-precision. (Since BASIC classifies all variables as single precision initially, you need to use DEFSNG only if one of the other DEF statements is used.)

Example:

```
DEFSTR L
```

BASIC classifies all variables that start with L as string variables. After this statement, the variables

```
L          LP          LAST
```

can hold string values only.

As with constants, you can override the type of a variable name by adding a type declaration tag at the end.

For example:

```
I%        FT%       NUM%     COUNTER%
```

are all integer variables, **regardless** of what attributes have been assigned to the letters I, F, N, and C.

```
T!        RY!       QUAN!    PERCENT!
```

are all single-precision variables, **regardless** of what attributes have been assigned to the letters T, R, Q, and P.

```
X#        RR#       PREV#    LSTNUM#
```

are all double-precision variables, **regardless** of what attributes have been assigned to the letters X, R, P, and L.

```
Q$        CA$       WRD$     ENTRY$
```

are all string variables, **regardless** of what attributes have been assigned to the letters Q, C, W, and E.

Any variable name can represent four different variables. For example:

```
A5#       A5!       A5%      A5$
```

are all valid and **distinct** variable names.

## How BASIC Converts Numeric Data

A statement in your BASIC program may contain numbers with different degrees of precision. When BASIC evaluates the expression, all operands are converted to the same degree of precision, that of the most precise operand. The result of the arithmetic operation is also returned to this degree of precision.

Often your program might ask BASIC to assign one type of constant to a different type of variable. For example:

```
A% = 2.34
```

In this example, BASIC must first convert the single-precision constant 2.34 to an integer in order to assign it to the integer variable A%.

You might also want to convert one type of variable to a different type, such as:

```
A# = A%
A! = A#
A! = A%
```

## Single or double precision to integer type

BASIC rounds the fractional portion of the number.

**Note:** The original value must be in the range −32768 to 32768.

Examples

```
A% = 32766.7
```

assigns A% the value 32767.

```
A% = 2.5D3
```

assigns A% the value 2500.

```
A% = -123.45678901234578
```

assigns A% the value −123.

```
A% = -32768.5
```

produces an Overflow Error (out of integer range).

## Integer to single or double precision

BASIC appends a decimal point and zeroes to the right of the original value.

Examples

```
A# = 32767
```

Stores 32767.000000000000 in A#.

```
A! = -1234
```

Stores −1234.000 in A!.

## Double to single precision

BASIC rounds the number to seven significant digits.

Examples

```
A! = 1.234567890124567
```

stores 1.234568 in A!. However, the statement:

```
A! = 1.3333333333333333
```

stores 1.333333 in A!.

## Single to double precision

BASIC adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, the resulting value is accurate. For example:

```
A# = 1.5
```

stores 1.5000000000000 in A#, since 1.5 does have an exact binary representation.

However, for numbers that have no exact binary representation, the conversion creates an erroneous value. For example:

```
A# = 1.3
```

stores 1.299999952316284 in A#.

You should keep such conversions out of your programs because most fractional numbers do not have an exact binary representation. For example, when you assign a constant value to a double-precision variable, you can force the constant to be double precision:

```
A# = 1.3#      A# = 1.3D
```

both store 1.3 in A#.

**Here is a special technique** for converting a single precision value to double precision accurately. It is useful when the single-precision value is stored in a variable.

Convert the single-precision variable to a string with STR$; then convert the resultant string into a number with VAL.

That is, use:

VAL(STR$(single-precision variable))

For example, the following program

```
10 A! = 1.3
20 A# = A!
30 PRINT A#
```

prints a value of:

```
1.299999952316284
```

This program

```
10 A! = 1.3
20 A# = VAL(STR$(A!))
30 PRINT A#
```

prints a value of

```
1.3
```

The conversion in line 20 causes the value in A! to be stored accurately in double-precision variable A#.

## Illegal Conversions

BASIC cannot automatically convert numeric values to string, or vice versa. For example, the statements:

```
A$ = 1234
A% = "1234"
```

are illegal. They return a "Type mismatch" error. (Use STR$ and VAL to accomplish such conversions.)

# How BASIC Manipulates Data

BASIC has many fast methods to count, sort, test, and rearrange your data. These methods fall into two categories:

1. Operators
   a. numeric
   b. string
   c. relational
   d. logical
2. Functions

## Operators

An operator is a symbol or word that signifies some action to be taken on specified values. The data that the operations are performed on are called operands.

In general, an operator is used like this:

| operand-1 | operator | operand-2 |
|---|---|---|
| 6 | + | 2 |

The addition operator, plus (+), connects or relates its two operands, 6 and 2, to produce the result 8.

Operand-1 and -2 can be expressions.

A few operations take only one operand, and are used like this:

| operator | operand |
|---|---|
| – | 5 |

The negative operator, minus (–), acts on the single operand 5 to produce the result negative 5.

Neither 6 + 2, nor –5 can stand alone; they must be used in statements to be meaningful to BASIC. For example:

```
A = 6 + 2
PRINT -5
```

Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

based on the kinds of operands they require and the results they produce.

## Numeric Operators

Numeric operators are used in numeric expressions. Their operands must always be numeric, and the result they produce is one numeric data item. Unless otherwise stated, when BASIC evaluates the expression, all operands are converted to the same degree of precision, that of the most precise operand. The result of the arithmetic operation is also returned to this degree of precision.

There are seven numeric operators. Two of them, plus ( + ) and minus ( − ), are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

```
PRINT -77, +77
```

the sign operators − and + produce the values negative 77 and positive 77, respectively.

**Note:** When no sign operator appears in front of a numeric term, + is assumed.

The other numeric operators are binary; that is, they all take two operands.

These operators are, in order of precedence:

| | |
|---|---|
| ^ | Exponentiation |
| *, / | Multiplication, Division |
| \,MOD | Integer Division, Modulus Arithmetic |
| +,− | Addition, Subtraction |

### Exponentiation

The symbol ^ denotes exponentiation. It converts both its operands to single precision and returns a single-precision result.

Examples:

```
PRINT 2^3
```

prints 8. 2 * 2 * 2 is 8.

```
PRINT 6^.3
```

prints 6 to the .3 power.

### Multiplication

The asterisk (*) is the symbol for multiplication.

Examples:

```
PRINT 33 * 11%
```

performs integer multiplication and prints 363.

```
PRINT 33 * 11.1
```

performs single-precision multiplication and prints 366.3.

```
PRINT 12.345678901234567 * 11
```

performs double-precision multiplication and prints 135.8024679135802.

### Division

The slash (/) is the symbol for ordinary division.

Examples:

```
PRINT 3/4
```

performs single-precision division and prints 0.75.

```
PRINT 3.8/4
```

performs single-precision division and prints 0.95.

**PRINT 1358025679135802375/11**

performs double-precision division and prints 1.234568799214366D+16

**Integer Division**

The \ (backslash) is the symbol for integer division. Both operands are rounded to integers, and the result is truncated to an integer.

Examples:

**PRINT 10 \ 4**

prints 2.

**PRINT 68\ 6.99**

prints 9.

**Modulus Arithmetic**

MOD is the operator for modulus arithmetic. Both operands are rounded to integers. The result is the integer that is the remainder of an integer division.

Examples:

**PRINT 10 MOD 3**

prints 1. Ten divided by 3 is 3 with a remainder of 1.

**PRINT 68 MOD 6.99**

prints 5. 68 divided by seven is 3 with a remainder of 5.

**Addition**

The plus (+) is the symbol for addition.

Examples:

**PRINT 2 + 3**

performs integer addition and prints 5

**PRINT 3.1 + 3**

performs single-precision addition and prints 6.1

PRINT 1.2345678901234567 + 1

performs double-precision addition and prints 2.234567890123457.

**Subtraction**

The minus ( – ) is the symbol for subtraction.

Examples:

PRINT 33 – 11

performs integer subtraction and prints 22

PRINT 33 – 11.1

performs single-precision subtraction and prints 21.9

PRINT 12.345678901234567 – 11

performs double-precision subtraction and prints 1.34567890123457.

## String Operator

BASIC has a string operator ( + ) to concatenate (append) two strings into one. The concatenation symbol is used as part of a string expression. The operands are both strings, and the resulting value is one piece of string data.

The + operator appends the string on the right of the symbol to the string on the left of the symbol. For example:

PRINT "CATS" + "LOVE" + "MICE"

prints:

CATSLOVEMICE

Since BASIC does not allow one string to be longer than 255 characters, you get an error if your resulting string is too long.

## Relational Operators

Relational operators compare two numerical or two string expressions to form a relational expression. This expression reports whether the comparison you set up in your program is true or false. It returns a −1 if the relation is true; a 0 if it is false.

### Numeric Relations

This is the meaning of the operators when you use them to compare numeric expressions:

| Operator | Meaning |
|---|---|
| < | Less than |
| > | Greater than |
| = | Equal to |
| <> or >< | Not equal to |
| =< or >= | Less than or equal to |
| => or <= | Greater than or equal to |

Examples of true relational expressions:

1 < 2
2 <> 5
2 <= 5
2 <= 2
5 > 2
7 = 7

### String Relations

The relational operators for string expressions are the same as above, although their meanings are slightly different. Instead of comparing numerical magnitudes, the operators compare their ASCII sequence. This allows you to sort string data:

| Operator | Meaning |
|---|---|
| < | Precedes |
| > | Follows |
| >< or <> | Does not have the same precedence |
| <= | Precedes or has the same precedence |
| >= | Follows or has the same precedence |

BASIC compares the string expressions on a character-by-character basis. When it finds a non matching character, it checks to see which character has the lower ASCII code. The character with the lower ASCII code is the smaller (precedent) of the two strings.

Examples of true relational expressions:

"A" < "B"

The ASCII code for A is decimal 65; for B it's 66.

"CODE" < "COOL"

The ASCII code for O is 79; for D it's 68.

If while making the comparison, BASIC reaches the end of one string before finding non matching characters, the shorter string is the precedent. For example:

"TRAIL" < "TRAILER"

Leading and trailing blanks are significant. For example:

" A" < "A"

ASCII for the space character is 32; for A, it's 65.

"ABCD" < "ABCDE"

The string on the left is four characters long; the string on the right is five.

**How to Use Relational Expressions**

Normally, relational expressions are used as the test in an IF/THEN statement. For example:

```
IF A = 1 THEN PRINT "CORRECT"
```

BASIC tests to see if A is equal to 1. If it is, BASIC prints the message.

```
IF A$ < B$ THEN 50
```

if string A$ alphabetically precedes string B$, then the program branches to line 50.

```
IF R$ = "YES" THEN PRINT A$
```

if R$ equals YES then the message stored as A$ is printed.

You may also use relational expressions to return the true or false results of a test. For example:

```
PRINT 7 = 7
```

prints − 1 since the relation tested is true.

```
PRINT "A" > "B"
```

prints 0 because the relation tested is false.

## Logical Operators

Logical operators make logical comparisons. Normally, they are used in IF/THEN statements to make a logical test between two or more relations. For example:

```
IF A = 1    OR C = 2    THEN PRINT X
```

The logical operator, OR, compares the two relations A = 1 and C = 2.

Logical operators may also be used to make bit comparisons of two numeric expressions.

For this application, BASIC does a bit-by-bit comparison of the two operands, according to predefined rules for the specific operator.

**Note:** The operands are converted to integer type, stored internally as 16-bit, two's complement numbers. To understand the results of bit-by-bit comparisons, you need to keep this in mind.

The following table summarizes the action of Boolean operators in bit manipulation.

| Operator | Meaning of Operation | First Operand | Second Operand | Result |
|---|---|---|---|---|
| NOT | Result is opposite of bit. | 1 | | 0 |
| | | 0 | | 1 |
| AND | When both bits are 1, the results will be 1. Otherwise, the result will be 0. | 1 | 1 | 1 |
| | | 1 | 0 | 0 |
| | | 0 | 1 | 0 |
| | | 0 | 0 | 0 |
| OR | Result will be 1 unless both bits are 0. | 1 | 1 | 1 |
| | | 1 | 0 | 1 |
| | | 0 | 1 | 1 |
| | | 0 | 0 | 0 |
| XOR | When one of the bits is 1, the result is 1. Otherwise the result is 0. | 1 | 1 | 0 |
| | | 1 | 0 | 1 |
| | | 0 | 1 | 1 |
| | | 0 | 0 | 0 |
| EQV | When both bits are 1 or both bits are 0, the result is 1. | 1 | 1 | 1 |
| | | 1 | 0 | 0 |
| | | 0 | 1 | 0 |
| | | 0 | 0 | 1 |
| IMP | The result is 1 unless the first bit is 1 and the second bit is 0. | 1 | 1 | 1 |
| | | 1 | 0 | 0 |
| | | 0 | 1 | 1 |
| | | 0 | 0 | 1 |

## Hierarchy of Operators

When your expressions have multiple operators, BASIC performs the operations according to a well-defined hierarchy so that results are always predictable.

### Parentheses

When a complex expression includes parentheses, BASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. For example, the expression:

$$8 - (3 - 2)$$

is evaluated like this:

$$3 - 2 = 1$$
$$8 - 1 = 7$$

With nested parentheses, BASIC starts evaluating the innermost level first and works outward. For example:

$$4 * (2 - (3 - 4))$$

is evaluated like this:

$$3 - 4 = -1$$
$$2 - (-1) = 3$$
$$4 * 3 = 12$$

### Order of Operations

When evaluating a sequence of operations on the same level of parenthesis, BASIC uses a hierarchy to determine what operation to do first.

The two listings below show the hierarchy. Operators are in decreasing order of precedence and are executed as encountered **from left to right:**

For Numeric Operations:

| Operator | Description |
|---|---|
| ( ) | (Parentheses) |
| ^ | (Exponentiation) |
| +,− | (Unary sign operands [**not** addition and subtraction]) |
| *, / | (Multiplication and division) |
| \,MOD | (Integer Division and Modulus Arithmetic) |
| +,− | (Addition and subtraction) |
| <,>,=,<=,>=,<> | |
| NOT | |
| AND | |

OR
XOR
EQV
IMP

For String Operations:

+
<,>,=,<=,>=,<>

For example, in the line:

```
X * X + 5^2.8
```

BASIC finds the value of 5 to the 2.8 power. Next it multiplies X*X, and finally it adds that value to the value of 5 to the 2.8 power. If you want BASIC to perform the indicated operations in a different order, you must add parentheses. For example:

```
X * (X + 5)^2.8
```

BASIC adds the value X + 5 and raises that value to the power before it performs the multiplication.

Here's another example:

```
IF X = 0 OR Y > 0 AND Z = 1 THEN 255
```

The relational operators = and > have the highest precedence, so BASIC performs them first, one after the next, from left to right. Then the logical operations are performed. AND has a higher precedence than OR, so BASIC performs the AND operation before OR.

If the above line looks confusing because you can't remember which operator is precedent over which, then you can use parentheses to make the sequence obvious:

```
IF X = 0 OR ((Y > 0) AND (Z = 1)) THEN 255
```

## Functions

A function is a built-in sequence of operations that BASIC performs on data. BASIC functions make writing a BASIC routine unnecessary, and they operate faster than a routine would.

Examples:

**SQR (A + 6)**

BASIC computes the square root of (A + 6).

**MID$ (A$,3,2)**

BASIC returns a substring of the string A$, starting with the third character, with a length of 2.

BASIC functions are described in more detail in Chapter 7.

If the function returns numeric data, it is a numeric function and may be used in a numeric expression. If it returns string data, it is a string function and may be used in a string expression.

# How to Construct an Expression

Understanding how to construct an expression will help you put together powerful statements — instead of using many short ones. In this section we discuss the two kinds of expressions you may construct

- Simple
- Complex

as well as how to construct a function.

An expression is actually data. This is because once BASIC performs all the operations, it returns one data item. An expression may be string or numeric. It may be composed of:

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex.

## Simple Expressions

A **simple expression** consists of a single term: a constant, variable, or function. If it is a numeric term, it may be

preceded by an optional + or − sign or by the logical operator NOT.

For example:

+A    3.3    −5    SQR(8)

are all simple numeric expressions, since they consist of only one numeric term.

A$    STRING$ (20,A$)    "WORD"    "M"

are all simple string expressions, since they consist of only one string term.

## Complex Expressions

A *complex expression* consists of two or more terms (simple expressions) combined by operators. For example:

A-1    X+3.2−Y    1=1    A AND B
ABS(B)+LOG(2)

are all complex numeric expressions. Notice that you can use the relational expression (1=1) and the logical expression (A AND B) as a complex numeric expression since both actually return numeric data.

A$ + B$    "Z" + Z$    STRING$(10, "A") + "M"

are all examples of complex string expressions.

Most functions, except those functions returning system information, require that you input either or both of the following kinds of data:

- One or more numeric expressions
- One or more string expressions

If the data returned is a number, the function may be used as a term in a numeric expression. If the data is a string, the function may be used as a term in a string expression.

SIN(A)    STR$(X)    VAL(A)    LOG(.53)

are all examples of functions.

# Chapter 5

# Disk Files

You may want to store data on your disk for future use. To do this, you need to store the data in a "disk file." A disk file is an organized collection of related data. It may contain a mailing list, a personnel record, or almost any kind of information. This is the largest block of information on disk that you can address with a single command.

To transfer data from a BASIC program to a disk file, and vice-versa, the data must first go through a "buffer". This is an area in memory where data is accumulated for further processing.

With BASIC, you can create and access two types of disk files: sequential access or direct access.

## Sequential-Access Files

With a sequential-access file, you can only access data in the same order it was stored: sequentially. To read from or write to a particular record in the file, you must first read through all the records in the file until you get to the desired record.

Data is stored in a sequential file as ASCII characters. Therefore, it is ideal for storing variable length data without wasting space between data. However, it is limited in flexibility and speed.

The statements and functions used with sequential files are:

| | | | |
|---|---|---|---|
| OPEN | WRITE# | EOF | LOF |
| PRINT# | INPUT# | LOC | |
| PRINT# USING | LINE INPUT# | CLOSE | |

These statements and functions are discussed in more detail in Chapters 6 and 7.

# Creating a Sequential-Access File

1. To create the file, OPEN it in "O" (output) mode and assign it a buffer number (from 1 to 15).

### Example

```
OPEN "O", 1, "LIST.EMP"
OPEN "LIST.EMP" FOR OUTPUT AS 1
```

either of these forms of the syntax for the OPEN statement opens a sequential output file named LIST.EMP and gives buffer 1 access to this file.

2. To input data from the keyboard into one or more program variables, use either INPUT or LINE INPUT. (The difference between these two statements is that each recognizes a different set of "delimiters". Delimiters are characters that define where a data item begins or ends).

### Example

```
LINE INPUT, "NAME? "; N$
```

inputs data from the keyboard and stores it in variable N$.

3. To write data to the file, use the WRITE# statement (you can also use PRINT#, but make sure you delimit the data).

### Example

```
WRITE# 1, N$
```

writes variable N$ to the file, using buffer 1 (the buffer used to OPEN the file). Remember that data must go through a buffer before it can be written to a file.

4. To ensure that all the data was written to the file, use the CLOSE statement.

## Example

```
CLOSE 1
```

closes access to the file, using buffer 1 (the same buffer used to OPEN the file).

## Sample Program

```
10 OPEN "O", 1, "LIST.EMP"
20 LINE INPUT "NAME? ";N$
30 IF N$ = "DONE" THEN 60
40 WRITE# 1, N$
50 PRINT: GOTO 20
60 CLOSE 1
RUN
```

**Note:** The file "LIST.EMP" stores the data you input through the aid of the program, not the program itself (the program manipulates data). To save the program above, you must assign it a name and use the SAVE command (refer to Chapter 1).

## Example

```
SAVE "PAYROLL.BAS"
```

saves the program under the name "PAYROLL.BAS".

**Note:** Every time you modify a program, you must SAVE it again (you can use the same name); otherwise, the original program remains on disk, without your latest corrections. If the filename is eight characters or less and you do not include an extension in the file name, BASIC appends the extension ".BAS" when you use the SAVE, MERGE, LOAD, and RUN statements.

5. To access data in the file, reOPEN it in the "I" (input) mode.

### Example

```
OPEN "LIST.EMP" FOR INPUT AS 1
```

OPENs the file named LIST.EMP for sequential input, using buffer 1.

6. To read data from the file and assign it to program variables, use either INPUT# or LINE INPUT#.

### Examples

```
INPUT# 1, N$
```

reads a string item into N$, using buffer 1 (the buffer used when the file was OPENed).

```
LINE INPUT# 1, N$
```

reads an entire line of data into N$, using buffer 1.

INPUT# and LINE INPUT# each recognize a different set of "delimiters" for reading data from the file. Delimiters are characters that define the beginning or end of a data item. See Chapter 7 for a detailed explanation of these statements.

### Sample Program

```
 10 OPEN "I", 1, "LIST.EMP"
 20 IF EOF(1), THEN 100
 30 INPUT# 1, N$
 40 PRINT N$
 50 GOTO 20
100 CLOSE
```

## Updating a Sequential-Access File

1. To add data to the file, OPEN it in "A" (append) mode.

```
OPEN "A", 1, "LIST.EMP"
```

opens the file LIST.EMP so that it can be extended. The data you enter is appended to LIST.EMP.

2. To enter new data to the file, follow the same procedure as for entering data in "O" mode.

The following program illustrates this technique. It builds on the file we previously created under the name LIST.EMP.

> **Note:** Read through the entire program first. If you encounter BASIC keywords (statements or functions) that are unfamiliar to you, refer to Chapter 7 for their definitions.

```
NEW
10 OPEN "A", 1, "LIST.EMP"
20 LINE INPUT "TYPE A NEW NAME OR PRESS
   <N>"; N$
30 IF N$ = "N" THEN 60
40 WRITE# 1, N$
50 GOTO 20
60 CLOSE
```

If you want the program to print on your display the information stored in the updated file, add the following lines:

```
  70 OPEN "LIST.EMP" FOR INPUT AS 1
  80 IF EOF(1) THEN 2000
  90 INPUT# 1, N$
 100 PRINT N$
 110 GOTO 80
2000 CLOSE
RUN
```

After you RUN this program, SAVE it.

## Example

```
SAVE "PAYROLL2.BAS"        'saves the new
                            program
```

# Direct-Access Files

With a direct-access file, you can access data anywhere on disk. It is not necessary to read through all the information, as with a sequential-access file. This is possible because in a direct-access file, information is stored and accessed in distinct units called "records". Each record is numbered.

Creating and accessing direct-access files requires more program steps than sequential-access files. However, direct-access files are more flexible and easier to update.

One important note: BASIC allocates space for records in numeric order. That is, if the first record you write to the file is number 200, BASIC allocates space for records 0 through 199 before storing record 200 in the file.

The maximum number of logical records is 65,535. Each record may contain between 1 and 128 bytes.

The statements and functions used with direct-access files are:

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

These statements and functions are discussed in more detail in Chapters 6 and 7.

## Creating a Direct-Access File

1. To create the file, OPEN it for random access in "R" mode.

### Example

```
OPEN, "R", 1, "LISTING.BAS", 32
```

opens the file named "LISTING.BAS", gives buffer 1 direct access to the file, and sets the record length to 32 bytes. (If you omit the record length, the default is 128 bytes). Remember that data is passed to and from disk in records.

2. Use the FIELD statement to allocate space in the buffer for the variables that you write to the file. This is necessary because you must place the entire record into the buffer before putting it into the disk file.

### Example

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

allocates the first 20 positions in buffer 1 to string variable N$, the next four positions to A$, and the next eight positions to P$. N$, A$ and P$ are now "field names".

3. To move data into the buffer, use the LSET statement. Numeric values must be converted into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.

### Example

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

4. To write data from the buffer to a record (within a direct-access disk file), use the PUT statement.

```
PUT 1, CODE%
```

writes the data from buffer 1 to a record with the number CODE%. (The percentage sign at the end of a variable specifies that it is an integer variable.)

The following program writes information to a direct-access file:

```
10 OPEN "LISTING.BAS" AS 1 LEN = 32
20 FIELD 1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE, 0 TO END"; CODE%
40 IF CODE% = 0 THEN 130
50 INPUT "NAME"; X$
60 INPUT "AMOUNT"; AMT
70 INPUT "PHONE"; TEL$
80 LSET N$ = X$
90 LSET A$ = MKS$(AMT)
100 LSET P$ = TEL$
110 PUT 1, CODE%
120 GOTO 30
130 CLOSE 1
```

The two-digit code that you enter in line 30 becomes a record number. That record stores the name(s), amount(s) and phone number(s) you enter when lines 50, 60 and 70 are executed. The record is written to the file when BASIC executes the PUT statement in line 110.

After typing this program, SAVE it and RUN it. Then, enter the following data:

```
2-DIGIT CODE, 0 TO END? 20
NAME? SMITH
AMOUNT? 34.55
PHONE? 567-9000
2-DIGIT CODE, 0 TO END? 0
```

BASIC stored SMITH, 34.55, and 567-9000 in record 20 of file LISTING.

## Accessing a Direct-Access File

1. OPEN the file in "R" mode.

### Example

```
OPEN "R", 1,"LISTING.BAS",32
```

2. Use the FIELD statement to allocate space in the buffer for the variables that are read from the file.

### Example

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

3. Before you use the GET statement to read the record, set a variable in your program equal to the record size used in the OPEN statement. This provides a way for you to check that you are not attempting to access a record that is not in your file. LOF returns the length of the file in bytes. The total number of bytes in the file can't be less than the requested record number multiplied by the record size. An attempt to access a record number greater than the largest record number in the file results in an "Input Past End" error.

### Example

```
RECSIZE% = 32
IF (CODE% * RECSIZE%) > LOF(1) THEN 1000
```

4. Use the GET statement to read the desired record from a direct disk file into a buffer.

### Example

```
GET 1, CODE%
```

gets the record numbered CODE% and reads it into buffer 1.

5. Convert string values back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

### Example

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

The program may now access the data in the buffer.

The following program accesses the direct-access file "LISTING.BAS" (created with the previous program). When BASIC executes line 30, enter any *valid* record number from "LISTING.BAS". This program will print the contents of that record.

```
10 OPEN "R", 1, "LISTING.BAS", 32
20 FIELD 1,20 AS N$,4 AS A$,8 AS P$
30 RECSIZE% = 32
40 INPUT "2-DIGIT CODE, 0 TO END"; CODE%
50 IF CODE% = 0 OR (CODE% * RECSIZE%) >
   LOF(1) THEN 1000
60 GET #1, CODE%
70 PRINT N$
80 PRINT USING "$$#.##"; CVS(A$)
90 PRINT P$: PRINT
100 GOTO 40
1000 CLOSE 1
```

After typing this program, SAVE it and RUN it. When BASIC asks you to enter a 2-digit code, enter 20 (the record we created through the previous program). Your display should show:

```
2-DIGIT CODE, 0 TO END?
SMITH
$34.55
567-9000
```

If you entered a record number which is not a part of "LISTING.BAS", your display would show:

```
$0.00
```

If you wanted to go back and update "LISTING.BAS", simply LOAD the previous program (the one that created "LISTING.BAS") and RUN it.

# Chapter 6

# Introduction To BASIC Keywords

BASIC is made up of keywords. These keywords instruct the computer to perform certain operations.

Chapter 7 describes all of BASIC's keywords. This chapter explains the format used in Chapter 7. It also introduces you to BASIC's two types of keywords: statements and functions.

# Format for Chapter 7

| **Keyword** | **Communications**<br>**Graphics** | **Statement**<br>**Function** |
|---|---|---|

## Syntax *parameter(s)* or (*argument(s)*)

Brief definition of keyword.

Detailed definition of keyword.

## Example(s)

## Sample Program(s)

Communications indicates that the keyword performs a specific operation when used with the RS-232C asynchronous communications adapter. Graphics indicates that the keyword has a specific operation when used with either the black and white graphics board (catalog number 26-5140) or the color graphics board (catalog number 26-5141). The RS-232C asynchronous communications adapter is standard on your Model 2000. You must purchase the optional graphics boards to use the graphics commands.

This format varies slightly, depending on the complexity of each keyword. For instance, some keywords are used alone (without parameters or arguments). Others have several possible syntaxes. As a general rule, definitions for statements are longer than definitions for functions. That is because a statement is a complete instruction to BASIC, while a function is a built-in subroutine which may only be used as part of a statement.

Some keywords have several sample programs, others don't have any at all. We added programs to illustrate useful applications which may not be readily apparent. Remember that this manual is to be used as a reference, not a tutorial on how to program in BASIC.

**Important Note**: BASIC for MS-DOS requires that keywords be delimited by spaces. This means that you must leave a space between a keyword and any variables, constants or other keywords. The only exceptions to this rule are characters which are shown as part of the syntax of the keyword.

For example, if you type:

```
DELETE.
```

BASIC returns a "Syntax error." You must leave a blank space between the word DELETE and the period.

For a definition of the terms and notation used in Chapter 7, see page 1-2 of the Introduction.

# Statements

A program is made up of lines; each line contains one or more statements. A statement tells the computer to perform some operation when that particular line is executed. For example,

```
100 STOP
```

tells the computer to stop executing the program when it reaches line 100.

**Statements for assigning values to variables and defining memory space:**

| | |
|---|---|
| CLEAR | clears all variables, allocates memory and stack space. |
| COMMON | passes variables to a CHAINed program. |
| DATA | stores data in your program so that you may assign it to a variable. |
| DEFDBL | defines variables as double precision. |

| | |
|---|---|
| DEF FN | defines a function according to your specifications. |
| DEFINT | defines variables as integers. |
| DEF SEG | defines the current segment address. |
| DEFSNG | defines variables as single precision |
| DEFSTR | defines variables as strings. |
| DEF USR | defines the offset of the entry point for USR routines |
| DIM | dimensions an array. |
| ERASE | erases an array. |
| LET | assigns a value to a variable (the keyword LET may be omitted). |
| MID$ | replaces a portion of a string. |
| OPTION BASE | declares the minimum value for array subscripts. |
| RANDOMIZE | reseeds the random number generator. |
| READ | reads data stored in the DATA statement and assigns it to a variable. |
| RESTORE | restores the DATA pointer. |
| SWAP | exchanges the values of variables. |

**Statements for altering program sequence:**

| | |
|---|---|
| CALL | calls an assembly language subroutine. |
| CHAIN | loads another program and passes variables to the current program. |
| COM(1) ON | enables communication trapping. |
| END | ends a program. |
| FOR/NEXT | establishes a program loop. |
| GOSUB | transfers program control to the subroutine |
| GOTO | transfers program control to the specified line number. |
| IF . . . THEN . . . ELSE | evaluates an expression and performs an operation if conditions are met. |
| KEY(n) ON | enables key trapping. |
| ON COM(1) GOSUB | branches to a subroutine when activity occurs on the communication channel. |

| | |
|---|---|
| ON KEY . . . GOSUB | branches to a subroutine when a specific key is pressed. |
| ON . . . GOSUB | evaluates an expression and branches to a subroutine. |
| ON . . . GOTO | evaluates an expression and branches to another program line. |
| ON STRIG...GOSUB | branches to a subroutine when you press a mouse button. |
| RETURN | returns from a subroutine to the calling program. |
| STOP | stops program execution. |
| STRIG ON | enables the STRIG function. |
| STRIG( ) ON | enables mouse trapping. |
| WHILE . . . WEND | executes statements in a loop as long as a given condition is true. |
| WAIT | suspends program execution while monitoring the status of a machine input port. |

**Statements for storing and accessing data on disk:**

| | |
|---|---|
| CLOSE | closes access to a disk file. |
| FIELD | organizes a direct-access buffer. |
| GET | gets a record from a direct-access file, or transfers a specific number of bytes from a communication file. |
| INPUT# | inputs data from a disk file. |
| LINE INPUT# | inputs an entire line from a disk file. |
| LSET | moves data (and left-justifies it) to a field in a direct-access file buffer. |
| OPEN | opens a disk file. |
| OPEN "COM | opens a communication file. |
| PRINT# | writes data to a sequential disk file. |
| PRINT# USING | writes data to a disk file using the specified format. |

| | |
|---|---|
| PUT | puts a record into a direct-access file or transfers a number of bytes to a communication file. |
| RESET | closes all open files on all diskettes. |
| RSET | moves data (and right-justifies it) to a field in a direct-access file buffer. |
| WRITE# | writes data to a sequential file. |

**Statements for debugging a program:**

| | |
|---|---|
| CONT | continues program execution. |
| ERL | returns the line number where an error occurred. |
| ERR | returns an error code after an error. |
| ERROR | simulates the specified error. |
| ON ERROR GOTO | sets up an error-trapping routine. |
| REM | inserts a remark line in a program. |
| RESUME | terminates an error-handling routine. |
| TROFF | turns the tracer off. |
| TRON | turns the tracer on. |

**Statements for inputting or outputting data to the video display or the line printer:**

| | |
|---|---|
| CIRCLE | draws an ellipse with a center and a radius on the display. |
| CLS | clears the display. |
| COLOR | to select foreground, background, and border display colors. |
| DRAW | draws images on the display. |
| GET | transfers graphic images from memory to the display. |
| INPUT | inputs data from the keyboard. |
| LINE | draws a line on the display. |
| LINE INPUT | inputs an entire line from the keyboard. |
| LIST | lists a program to the display or line printer. |

| | |
|---|---|
| LLIST | prints a program on the line printer. |
| LOCATE | positions the cursor on the screen. |
| LPRINT | prints data at the line printer. |
| PAINT | fills in an area of the screen with a selected color. |
| PRESET | draws a point in color at a specified position on the screen. |
| PRINT | lists data to the display. |
| PRINT USING | lists data to the display in a specific format. |
| PSET | draws a point on the screen at a specified position. |
| PUT | transfers graphic images from the display to memory. |
| SCREEN | sets the screen attributes (text, medium- or high-resolution) to be used by subsequent statements. |
| TAB | positions the cursor or the print head at a specified position. |
| WIDTH | sets the number of characters per line for the screen or line printer. |
| WRITE | prints data on the display. |

**Statements for performing system functions or entering other modes of operation:**

| | |
|---|---|
| AUTO | automatically numbers program lines. |
| BEEP | produces a sound from the computer speaker. |
| BLOAD | loads a memory image file from disk. |
| BSAVE | saves a memory image file to disk. |
| DELETE | erases program lines from memory. |
| EDIT | edits program lines. |
| KILL | deletes a disk file. |
| LOAD | loads a program from disk. |
| MERGE | merges a disk program with a resident program. |

| | |
|---|---|
| NAME | renames a disk file. |
| NEW | erases a program from RAM. |
| OUT | sends a byte to a machine output port. |
| POKE | writes a byte into a memory location. |
| PLAY | produces musical notes. |
| RENUM | renumbers a program. |
| RUN | executes a program. |
| SAVE | saves a program on disk. |
| SOUND | generates a specific tone for a specified length of time. |
| SYSTEM | returns to MS-DOS. |

# Functions

A function is a built-in subroutine. It may only be used as part of a statement.

Most BASIC functions return numeric or string data by performing certain built-in routines. Special print functions are used to control the video display.

**Numeric Functions (return a number):**

| | |
|---|---|
| ABS | computes the absolute value. |
| ASC | returns the ASCII code. |
| ATN | computes the arctangent. |
| CDBL | converts to double precision. |
| CINT | returns the largest integer not greater than the parameter. |
| COS | computes the cosine. |
| CSNG | converts to single precision |
| EXP | computes the natural exponential. |
| FIX | truncates to whole number. |
| FRE | returns the number of bytes in memory not being used. |
| INSTR | searches for a specified string. |
| INP | returns the byte read from a port. |
| INT | returns the largest whole number not greater than the argument. |

| | |
|---|---|
| LEN | returns the length of the string. |
| LOG | computes the natural logarithm. |
| LPOS | returns the position of the print head in the line printer buffer. |
| PEEK | returns a byte from a memory location. |
| RND | returns a pseudorandom number. |
| SGN | returns the sign. |
| SIN | calculates the sign. |
| SQR | calculates the square root. |
| TAN | computes the tangent. |
| USR | calls an assembly-language subroutine. |
| VAL | returns the numeric value of a string. |
| VARPTR | returns an offset for a variable or buffer. |

**String Functions (return a string value):**

| | |
|---|---|
| CHR$ | returns the specified character |
| DATE$ | sets or returns today's date. |
| ERRS$ | returns the latest error number and message. |
| HEX$ | converts a decimal value to a hexadecimal string. |
| LEFT$ | returns the left portion of a string. |
| MID$ | returns the mid-portion of a string. |
| OCT$ | converts a decimal value to an octal string. |
| RIGHT$ | returns the right portion of a string. |
| SPACE$ | returns a string of spaces. |
| STR$ | converts to string type. |
| STRING$ | returns a string of characters. |
| TIME$ | sets or returns the time. |

**Input/Output Functions (perform input/output to the keyboard, display, line printer or disk files):**

| | |
|---|---|
| CVD | restores data from a direct disk file to double precision. |

| | |
|---|---|
| CVI | restores data from a direct disk file to integer. |
| CVS | restores data from a direct disk file to single precision. |
| CRSLIN | returns the current row position of the cursor. |
| EOF | checks for end-of-file. |
| FILES | displays the names of the files on a diskette. |
| INKEY$ | returns the keyboard character. |
| INPUT# | inputs a string of characters from a sequential disk file. |
| INPUT$ | returns a string of characters from the keyboard. |
| KEY | assigns or displays the current function key soft values. |
| LOC | returns the current disk file record number. |
| LOF | returns the total number of bytes in a disk file or the amount of free space in a communication file input queue. |
| MKI$ | converts an integer value to a string for writing it to a direct-access disk file. |
| MKS$ | converts a single-precision number to a string for writing it to a direct-access file. |
| MKD$ | converts a double-precision value to a string for writing it to a direct-access file. |
| POS | returns the cursor column position on the display. |
| SCREEN | returns the ASCII code for the character stored at a specific position on the screen. |
| SPC | prints spaces to the display. |
| STICK | returns the number of points moved along the coordinates. |
| STRIG | returns the status of the mouse buttons. |

# Introduction to Graphics

Interpreter BASIC for your Model 2000 includes many new commands to display text and graphic images in black and white and in color. Which of these commands you can use and how you can use them depends on the graphics options you have.

In Chapter 7 the word Graphics is printed at the top of the pages on which there are statements that require a graphics option. The graphics commands that you can use to draw graphic images and perform animation are CIRCLE, DRAW, LINE, GET/Graphics, PAINT, POINT, PSET, PRESET, and PUT/Graphics. If you are using either of the color graphic options, you may also use the COLOR/Graphics, PALETTE, and PALETTE USING statements to draw the images in color.

## Graphics Options

You must have one of three graphics options to use any of the graphics commands. Each option provides a different degree of resolution. Resolution is the number of points on the screen. The greater number of points, the sharper the image.

In addition, each option controls the number of colors that can display on the screen at one time. BASIC provides fifteen colors. However, you may only use a certain number of colors at one time. The number of colors is also determined by the graphics option you have.

## Medium Resolution Color Graphics Option

To use the Medium Resolution Color Graphics Option you must have a color television set and the TV/Joystick Option (catalog number 26-5143).

The Medium Resolution Color Graphics Option provides 320 x 200 points in four colors. That means there are 320 vertical columns of 200 points each or 200 horizontal rows of 320 points each. Each horizontal row of points is numbered 0-320. Each vertical column of points is numbered 0-200. The point in the upper left corner of the display is 0,0. The point in the lower left corner of the display is 319,199.

With the Medium Resolution Color Graphics option you may display text or graphic images in 4 colors at one time. To display text, select Screen Mode 0. You can display 24 lines of 40 characters each. You can use 4 COLOR/Text statements to choose 4 of 15 colors to display letters, numbers, and special characters.

To use the graphics statements to draw graphic images and perform animation, select Screen Mode 1. You can specify any one point on the screen with the graphic statements. You can select the colors for the graphic images with the COLOR/Graphic, PALETTE, and PALETTE USING statements.

## High Resolution Monochrome Option

To use the Medium Resolution Monochrome Option you must have VM-1 Monochrome Monitor and a Monochrome Graphics Option Board (catalog # 26-5140).

The High Resolution Monochrome Graphics Option provides 640 vertical points. The number of horizontal points depends on the screen mode you select with the SCREEN statement. You may have either 200 or 400 horizontal points. Remember, the more points, the sharper the image.

Screen Mode 2 selects 640 x 200 points. That means there are 640 vertical columns of 200 points each or 200 horizontal rows of 640 points each. Each horizontal row of points is numbered 0-640. Each vertical column of points is numbered 0-200. The point in the upper left corner of the display is 0,0. The point in the lower left corner of the display is 639,199.

Screen Mode 4 selects 640 x 400 points. That means that there are 640 vertical columns of 400 points each or 400 horizontal rows of 640 points each. Each horizontal row of points is numbered 0-640. Each vertical column of points is numbered 0-400. The point in the upper left corner of the display is 0,0. The point in the lower left corner of the display is 639,399.

With the High Resolution Monochrome Graphics Option you can display text or graphic images in black and white. There are two shades of white, white and high-intensity white (brighter white.)

To display text, select SCREEN 0. You can use the COLOR/Text statement to create reverse image, invisible, highlighted, and underscored characters. You can display 24 rows of 40 characters or 24 rows of 80 characters by setting the screen width with the WIDTH statement.

To draw graphic images and perform animation select Screen Modes 2 or 4. The only difference between the two screen modes is the degree of resolution. By changing the parameters in the COLOR/Text statement, you can create reverse image, invisible, and highlighted graphic images.

## High Resolution Color Graphics Option

To use the High Resolution Color Graphics Option you must have a CM-1 Color Monitor (catalog # 26-5112), a Monochrome Graphics Option Board (catalog # 26-5140), and a Color Graphics Option kit (catalog # 26-5141).

The High Resolution Color Graphics Option provides 640 x 400 points in 8 colors. That means that there are 640 vertical columns of 400 points each or 400 horizontal rows of 640 points each. Each horizontal row of points is numbered 0-640. Each vertical column of points is numbered 0-400. The point in the upper left corner of the display is 0,0. The point in the lower left corner of the display is 639,399.

With the High Resolution Color Graphics Option you may display text or graphic images in black and white or in 8 of 15 colors. The screen mode you select with the SCREEN statement determines the color and resolution of the graphic images. You may select any of the 5 screen modes, 0, 1, 2, 3, or 4 and use any of the graphic options described.

To display text, select Screen Mode 0. You can display 24 lines of 40 characters each or 24 lines of 80 characters each. You can use 8 COLOR/Text statements to choose 8 of 15 colors to display letters, numbers, and special characters.

To use the graphics statements to draw colored graphic images and perform animation, select Screen Mode 1 or Screen Mode 3. Again, the only difference between the two screen modes is the degree of resolution.

If you select Screen Mode 1, you may specify horizontal coordinates in the range 0 to 320 and vertical coordinates in the range 0 to 199. If you select Screen Mode 3, you may specify horizontal coordinates in the range 0 to 640 and vertical coordinates in the range 0 to 400.

With either Screen Modes 1 or 3, you can display 8 of the 15 colors at one time. You can select the colors for the graphic images with the COLOR/Graphic, PALETTE, and PALETTE USING statements.

If you select Screen Modes 2 or 4, you can display graphic images in black and white. There are two shades of white, white and high-intensity white (brighter white).

The only difference between the two screen modes is the degree of resolution. By changing the parameters in the COLOR/Text statement, you can create reverse image, invisible, and highlighted graphic images.

## Specifying Coordinates

To draw your graphic images on the display, you must tell BASIC where to put the image on the screen. To do this, you must specify horizontal and vertical point numbers for the point you want to draw.

The horizontal and vertical point numbers are known as the coordinates. Coordinates are expressed as x-coordinate, y-coordinate. The x-coordinate is the horizontal point number, and the y-coordinate is the vertical point number. When you specify coordinates in a statement, separate them with a comma. Specified actual coordinates are called absolute coordinates.

You may also specify relative coordinates in some graphics commands. In this case, you specify offsets from the last graphics point referenced. An offset from the last graphics point referenced is a number of points away from the last point you drew. For example, if you use the CIRCLE statement to draw a CIRCLE, the last point BASIC draws is the center of the circle. If you then execute a LINE command and specify offsets rather than absolute coordinates, BASIC draws the line offset points away from the center of the circle.

You may specify positive or negative values for offsets. If you specify a negative value, BASIC subtracts offset from the coordinate of the last point referenced. If you specify positive values, BASIC adds offset to the coordinate of the last point referenced.

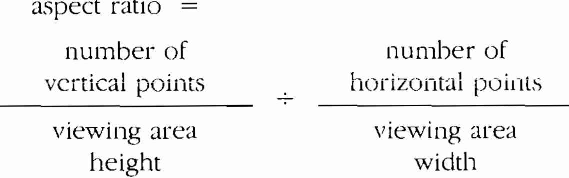
## Aspect Ratio

As you can see by our discussion of graphic options, there are more horizontal points than vertical points. In Screen Modes 1, 2, 3, and 4, the number of horizontal points in an inch is greater than the number of vertical points in an inch because the horizontal points are closer together than the vertical points. Aspect ratio is the realtionship between the number of points in a vertical inch to the number of points in a horizontal inch.

To calculate the screen aspect ratio, you must know the dimensions (height and width) of the viewing area of your monitor.

**Note:** The viewing area is that portion of your screen on which images are displayed. It may be smaller than the screen itself. Calculate the aspect ratio according to the following formula:

aspect ratio =



## Screen Mode 1

The standard viewing area has a width to height ratio of 4 to 3. This means that your monitor is 1 ⅓ times as wide as it high (regardless of the actual dimensions). For example, a viewing area that is 8 inches wide and 6 inches high has a width to height ratio of 8 to 6, which is the same as 4 to 3.

In Screen Mode 1, there are 320 horizontal points and 200 vertical points. To calculate the aspect ratio, substitute the actual values in the above formula.

$$200/6 \div 320/8 = 5/6$$

To calculate the number of points per inch in each direction, divide the total number of horizontal points by the width in inches and the total number of vertical points by the height in inches. There are 40 vertical points per inch and 33 horizontal points per inch on our example monitor.

Remember, to calculate points per inch and aspect ratio for your viewing area, you need to know actual dimensions and substitute those in the formula.

## Screen Mode 2

In Screen Mode 2 there are 640 horizontal points and 200 vertical points. The viewing area is 10 inches wide and 7 inches high. Substituting actual values in the formula gives the following equation:

$$200/7 \div 640/10 = 7/8$$

The aspect ratio is ⅞. There are 28 vertical points per inch and 64 horizontal points per inch.

## Screen Modes 3 and 4

In Screen Modes 3 and 4 there are 640 horizontal points and 400 vertical points. The viewing area is 10 inches wide and 7 inches high. Substituting actual values in the formula give the following equation:

$$400/7 \div 640/10 = 25/28$$

The aspect ratio is $25/28$. There are 57 vertical points per inch and 64 horizontal points per inch.

The aspect ratio for the current screen mode is important when using the graphics commands CIRCLE, DRAW, and LINE. Keep in mind that the number of horizontal points is not as long as the same number of vertical points. Therefore, if you try to draw a square, the perimeter of the square must contain more horizontal points than vertical points.

The CIRCLE statement compensates the difference in points per inch by letting you specify the aspect ratio. First, CIRCLE computes the x and y coordinates for each point on the ellipse. If the aspect ratio you specify is less than one, CIRCLE recomputes the y coordinates by multiplying the original y coordinates by the aspect ratio. If the aspect ratio you specify is larger than one, CIRCLE recomputes the x coordinates by multiplying the original x coordinates by aspect ratio.

You cannot specify an aspect ratio with the DRAW and LINE statements. You must compensate for the difference in points per inch yourself. When specifying the coordinates with the LINE and DRAW statements, keep in mind the aspect ratio for the current screen and adjust the coordinates so that the resulting image is what you intended.

Note also that because there is a difference in points per inch among the four different screen modes, images that specify the same coordinates do not look the same in different modes. For example, if you draw a vertical line in Screen Mode 2 with this statement

```
LINE (320,100)-(320,199)
```

the vertical line goes from the center of the screen to the bottom of the screen. However, if you use the same coordinates in Screen Mode 4, the center of the vertical line is in the center of the display and the line extends the same distance up from the center as down. The line does not extend to the bottom or to the top of the display.

# Chapter 7

# Statements And Functions

# ABS — Function

## ABS(*number*)

Computes the absolute value of *number*.

ABS returns the absolute value of the argument, that is, the magnitude of the number without respect to its sign.

If *number* is greater than or equal to zero, ABS(*number*) = *number*. If *number* is less than zero, ABS(*negative number*) = *number*.

## Example

```
X = ABS(Y)
```

computes the absolute value of Y and assigns it to X.

## Sample Program

```
100 INPUT "WHAT'S THE TEMPERATURE
    OUTSIDE (DEGREES F)"; TEMP
110 IF TEMP < 0 THEN PRINT "THAT'S"
     ABS(TEMP) "BELOW ZERO! BRR!": END
120 IF TEMP = 0 THEN PRINT "ZERO DEGREES!
     MITE COLD!": END
130 PRINT TEMP "DEGREES ABOVE ZERO?
     BALMY!": END
```

# ASC Function

## ASC(*string*)

Returns the ASCII code for the first character of *string*.

The value is returned as a decimal number. If *string* is null, an "Illegal function call" error occurs.

## Example

```
PRINT ASC("A")
```

prints 65, the ASCII code for "A".

## Sample Program

ASC can be used to make sure a program is receiving proper input. Suppose you've written a program that requires the user to input hexadecimal digits 0-9, A-F. To make sure that only those characters are input, and exclude all other characters, you can insert the following routine.

```
100 INPUT "ENTER A HEXADECIMAL VALUE
    (0-9,A-F)";N$
110 A = ASC(N$)        'get ASCII code
120 IF A>47 AND A<58 OR A>64 AND A<71
    THEN PRINT "OK.": GOTO 100
130 PRINT "VALUE NOT OK." : GOTO 100
```

# ATN **Function**

## ATN(*number*)

Computes the arctangent of *number* in radians.

ATN returns the angle whose tangent is *number*. The result is always single precision, regardless of *number*'s numeric type.

To convert this value to degrees, multiply ATN(*number*) by 57.29578.

## Example

```
X = ATN(Y/3)
```

computes the arctangent of Y/3 and assigns the value to X.

# AUTO **Statement**

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

Automatically generates a line number every time you press (ENTER).

AUTO begins numbering at *line* and displays the next line number after adding *increment*. The default for both values is 10. A period ( . ) can be substituted for *line*. In this case, BASIC uses the current line number. If *line* is followed by a comma, but you omit increment, BASIC assumes the last increment specified in the last AUTO statement or the default value of 10.

If AUTO generates a line number that already exists in the program, it displays an asterisk after the number. To save the existing line, press (ENTER) immediately after the asterisk. AUTO then generates the next line number.

To turn off AUTO, press (BREAK). The current line is canceled and BASIC returns to command level.

## Examples

```
AUTO
```

generates lines 10, 20, 30, 40.

```
AUTO 100, 50
```

generates lines 100, 150, 200, 250 . . .

# BEEP

Produces a sound from the computer speaker.

The BEEP statement sounds the speaker at 800 Hz for 1/4 second. For information on how to control the frequency or length of the sound, see the SOUND statement.

A BEEP statement has the same effect as

```
PRINT CHR$(7).
```

## Example

```
IF X > 20 THEN BEEP
```

If the variable X is out of range, the computer warns the operator with a beep.

# BLOAD Statement

## BLOAD *filespec* [,*offset*]

Loads a memory image file into memory.

*filespec* is a string expression that contains the drive identifier and filename. The filename is required. If you omit the drive identifier, BASIC assumes the current drive.

*offset* is an integer in the range 0 to 65535. *offset* represents a location away from the beginning of a segment. BASIC determines the address to load at from the segment address given in the most recently executed DEF SEG statement and *offset*. See DEF SEG.

If you omit *offset*, BASIC assumes the *offset* specified at BSAVE and loads the file into the same location from which it was saved.

If you specify *offset*, BASIC assumes you want to BLOAD at an address other than the address from which the program was saved and uses the last known DEF SEG address. Unless you want to load the file into BASIC's data segment, you must execute a DEF SEG statement before the BLOAD statement. If you used the /M: switch when you loaded BASIC, BLOAD the file at that offset. If you do not execute a DEF SEG before BLOAD, and you did not use the /M: switch when you loaded BASIC, and you specify *offset*, the file is loaded at that offset from BASIC's data segment, destroying BASIC's workspace.

**Note:** BLOAD does not perform an address range check. It is possible to load a file anywhere in memory. Therefore, you must be careful not to load over BASIC or over the operating system.

A memory image file is a byte-for-byte copy of what was originally in memory. See BSAVE for information about saving memory image files. See the section "Interfacing Assembly Language Programs", in Appendix E for more information on loading assembly language programs.

You may specify any segment as the target or source for BLOAD or BSAVE. This is a useful way to save and redisplay screen images by saving from or loading to the screen buffer.

**NOTE:** You may type (BREAK) at any time during BLOAD or LOAD, between files, or after a time-out period. BASIC exits the search and returns to direct mode. Previous memory contents remain unchanged.

## Sample Programs

**Program 1**

```
10 'SAVE A 50 byte image of memory
20 DEF SEG = &H10
30 FOR I = 256 to 306
40 VLUE = PEEK (I)
50 LPRINT "AT ADDRESS   " ;I; "  WE HAVE A
   VALUE OF   "; VLUE
60 NEXT I
70 BSAVE "PROG1", 0, 50
80 PRINT "Now Run Program 2 to verify that the
   contents saved in the file Prog1 match those in the
   print out produced by this program."
```

**Program 2**

```
10 'Load a 50 byte file into memory and verify it
20 DEF SEG = &H10
30 BLOAD "PROG 1. BAS", 0
40 FOR I = 256 TO 306
50 VALUE = PEEK(I)
60 LPRINT "AT ADDRESS   ";I; "the loaded value is
     ";VALUE
70 NEXT I
```

Program 1 saves a memory image file and Program 2 reloads that file and prints it.

# BSAVE **Statement**

## BSAVE *filespec, offset, length*

Saves the contents of an area of memory as a disk file.

*filespec* is a string expression that may contain the drive identifier and filename. Filename is required. If you omit the drive identifier, BASIC assumes the current drive.

*offset* is an integer in the range 0 to 65535. *offset* represents a location away from the beginning of a segment. BASIC determines the address to start saving from by the segment address used in the most recently executed DEF SEG statement and *offset*.

*length* is an integer in the range 1 to 65535. This is the length in bytes of the memory image file to be saved.

You must specify filename, *offset*, and *length*. If you omit any of them, a "Bad File Name" error is issued and BASIC aborts the save.

A memory image file is a byte-for-byte copy of what is in memory. The BSAVE statement lets you save data or programs as memory image files on disk. BSAVE is often used for saving assembly language programs, but you can also use it to save data, programs written in other languages, or screen images.

Unless you want to BSAVE part of BASIC's workarea or you used the /M: switch when you loaded BASIC, you must execute a DEF SEG statement before the BSAVE statement, since BASIC uses the address given in the most recently executed DEF SEG statement for the save. See DEF SEG and the section "Interfacing with Assembly Language Subroutines", in Appendix E for more information.

See BLOAD for an example of how to save a memory image file.

# CALL variable [(parameter list)]

Transfers program control to an assembly-language subroutine stored at *variable*.

*Variable* contains the offset into the segment where the subroutine starts in memory. *Variable* may not be an array variable. Offset must be on a 16-byte boundary.

*Parameter list* contains the variables that are passed to the external subroutine.

A CALL statement with no parameters generates a simple 8086 "CALL" instruction. The corresponding subroutine should return with a simple "RET".

The CALL statement is the recommended method of interfacing assembly language programs with BASIC programs. Do not use the USR function unless you are running previously written BASIC programs that already contain USR statements.

When a CALL statement is executed, BASIC transfers control to the subroutine through the address given in the last DEF SEG statement and the segment offset specified by *variable*. See the section "Interfacing Assembly Language Subroutines" in Appendix E for more details.

**Note:** The number, type and length of the parameters in the calling program must match with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those subroutines written in assembly language.

When accessing parameters in a subroutine, remember that they are pointers to the actual arguments passed.

## Example

```
110 MYROUT = &H0000
120 DEF SEG = &H1700
130 CALL MYROUT(I,J,K)
```

The subroutine, MYROUT, begins at offset 0 in the segment that begins at hexadecimal 1700. The values of I, J, and K (which we assume were given elsewhere) are passed to that routine.

# CDBL **Function**

## CDBL (*number*)

Converts *number* to double precision.

CDBL returns a 17-digit value. This function may be useful if you want to force an operation to be performed in double precision, even though the operands are single precision or integers.

## Sample Program

```
210 A=454.67
220 PRINT A; CDBL(A)
RUN
454.67 454.6700134277344
Ok
```

# CHAIN Statement

## CHAIN [MERGE] *filespec* [,*line*] [,ALL] [,DELETE *line-line*]

Loads a BASIC program named *filespec*, chains it to a "main" program, and begins running it.

*Filespec* must have been saved in ASCII format before you can CHAIN it. To do this, use SAVE with the 'A' option.

*Line* is the first line to be run in the CHAINed program. If you omit *line*, BASIC begins execution at the first program line of the CHAINed program.

The ALL option passes every variable in the main program to the chained program. If you omit the ALL option, the main program must contain a COMMON statement to pass variables. If you are CHAINing subsequent programs (and passing variables), each new program must contain a COMMON statement.

The MERGE option "overlays" the lines of filespec with the main program. See MERGE to understand how BASIC overlays (merges) program lines.

The DELETE option deletes *lines* in the overlay so that you can MERGE in a new overlay.

## Examples

```
CHAIN "PROG2.BAS"
```

loads PROG2.BAS, chains it to the main program currently in memory, and begins executing it.

```
CHAIN "SUBPROG.BAS",,ALL
```

loads, chains and executes SUBPROG.BAS. The values of all the variables in the main program are passed to SUB-PROG.BAS.

## Sample Program 1

```
10 REM THIS PROGRAM DEMONSTRATES
   CHAINING USING COMMON TO PASS
   VARIABLES.
```

```
20 REM SAVE THIS MODULE ON DISK AS
   "PROG1.BAS" 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.BAS"
90 PRINT : PRINT B$(1): PRINT : PRINT B$(2):
   PRINT
100 END
```

Save this program as "PROG1.BAS", using the 'A' option (Type: SAVE *filespec*, A). Type NEW, then enter the following program.

```
10 REM THE STATEMENT "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 DISK AS
   "PROG2.BAS" 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.BAS",90
90 END
```

Save this program as "PROG2.BAS", using the 'A' option. Load PROG1.BAS and run it. Your screen should display:

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.BAS'.

Type NEW and this program:

## Sample Program 2

```
10 REM THIS PROGRAM DEMONSTRATES
   CHAINING USING THE MERGE AND ALL
   OPTIONS.
20 A$="MAINPROG.BAS"
30 CHAIN MERGE "OVRLAY1.BAS", 1000, ALL
40 END
```

Save this program as "MAINPROG.BAS", using the 'A' option. Enter NEW, then type:

```
1000 PRINT A$;" HAS CHAINED TO
     OVRLAY1.BAS."
1010 A$="OVRLAY1.BAS"
1020 B$="OVRLAY2.BAS"
1030 CHAIN MERGE "OVRLAY2.BAS", 1000, ALL ,
     DELETE 1020-1040
1040 END
```

Save this program as "OVRLAY1.BAS", using the 'A' option. Enter NEW, then type:

```
1000 PRINT A$; " HAS CHAINED TO ";B$;"."
1010 END
```

Save this program as "OVRLAY2.BAS", using the 'A' option. Load MAINPROG.BAS and run it. Your screen should display:

```
MAINPROG.BAS HAS CHAINED TO OVRLAY1.BAS.
OVRLAY1.BAS HAS CHAINED TO OVRLAY2.BAS.
```

## Note

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

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

When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

# CHR$ **Function**

## CHR$ (*code*)

Returns the character corresponding to an ASCII or control *code*.

This is the inverse of the ASC function. CHR$ is commonly used to send a special character to the display.

## Examples

```
PRINT CHR$(35)
```

prints the character corresponding to ASCII code 35 (the character is #).

## Sample Program

The following program lets you investigate the effect of printing codes 32 through 255 on the display. (Codes 0 – 31 represent certain control functions.)

```
100 CLS
110 INPUT "TYPE IN THE CODE (32-255)"; C
120 PRINT CHR$(C);
130 GOTO 110
```

# CINT **Function**

## CINT (*number*)

Converts *number* to integer representation.

CINT rounds the fractional portion of *number* to make it an integer.

For example, PRINT CINT(1.5) returns 2; PRINT CINT( − 1.5) returns − 2. The result is a two-byte integer.

## Sample Program

```
PRINT CINT(17.65)
18
Ok
```

# CIRCLE **Statement**

## CIRCLE [STEP] (*x-coordinate, y-coordinate*) *,radius* [*,color,start,end,aspect*]

Draws an ellipse with the specified center and radius.

*x-coordinate* is the x coordinate of the center of the circle. In Screen Mode 1, *xcoordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4 *x-coordinate* may be in the range 0 to 640.

*y-coordinate* is the y coordinate of the center of the circle. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4 *y-coordinate* may be in the range 0 to 400.

If you include the STEP option, the numbers you specify as coordinates are offsets from the most recent graphics point referenced. *x-coordinate* is the number of points in the horizontal direction, and *y-coordinate* is the number of points in the vertical direction. Precede the numbers with a plus (+) or a minus (−) sign to indicate the direction (up, down, left, or right) from the most recent point referenced. The plus sign indicates to add the number to the most recent coordinate (right or up), and the minus indicates subtract (left or down) the number from the most recent coordinate.

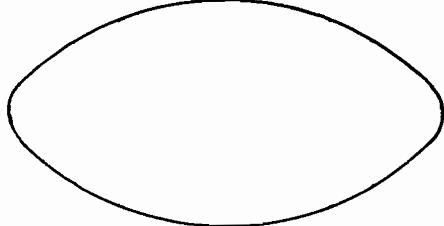
*color* indicates the color of the ellipse and must be a color number in the current palette. In Screen Mode 1, *color* may be in the range 0 to 3. In Screen Mode 3, *color* may be in the range 0 to 7. In Screen Modes 2 and 4, *color* may be either 0 or 1. If you omit *color* in Screen Modes 1 or 3, BASIC assumes color 3. If you omit *color* in Screen Modes 2 or 4, BASIC assumes white.

*radius* is the major axis of the ellipse.

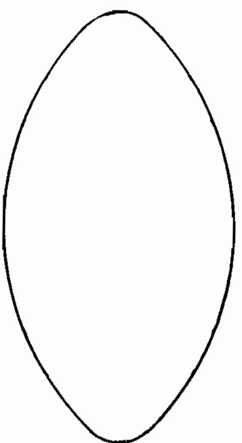
*start* and *end* are the beginning and ending angles in radians and must be in the range −6.283186 and 6.283186, or −2 * PI and 2 * PI. If you specify a negative *start* or *end* angle, the ellipse is connected to the center point with a line and the angles are treated as if they were positive.

*aspect* is the ratio of the *x* radius to the *y* radius in terms of coordinates. If *aspect* is less than one, *radius* is the *x* radius and is measured in points in the horizontal direction. If *aspect* is greater than one, *radius* is the *y* radius and is measured in points in the vertical direction. If you omit *aspect*, BASIC assumes 5/6 in screen mode 1, 25/56 in Screen Mode 2, and 25/28 in Screen Modes 3 and 4. When you use the default value, BASIC draws a circle.

To draw an ellipse that is wider than it is high, use an aspect ratio that is less than the default value for that screen mode. The smaller the aspect ratio you specify, the wider and shorter the ellipse is. For example, in Screen Mode 1, an aspect ratio of 1/2 gives you a wide, short ellipse like this:



To draw an ellipse that is higher than it is wide, use an aspect ratio that is larger than the default value for that screen mode. The larger the aspect ratio that you use, the taller and thinner the ellipse is. For example, in Screen Mode 1, an aspect ratio of 7/6 gives you a tall, thin ellipse like this:



See Chapter 6 "Introduction to Graphics" for more information on aspect ratio and specifying coordinates.

## Examples

```
10 SCREEN 1
20 CIRCLE (200,200),50
```

draws a circle with the center at point 200,200 and a radius of 50.

```
10 SCREEN 1
20 CIRCLE (160, 100), 60,,,,5/18
```

draws an ellipse with the center at point 160,100 and a radius of 60. Because the aspect ratio is less than the default value, the ellipse is wider than it is high.

# CLEAR **Statement**

## CLEAR [,*memory location*] [,*stack space*]

Clears the value of all variables and CLOSEs all open files.

*Memory location* must be an integer. It specifies the highest memory location available for BASIC. The default is the current top of memory (as specified with the /M: switch when BASIC was loaded). This option is useful if you will be loading a machine-language subroutine, since it prevents BASIC from using that memory area.

*Stack space* must also be an integer. This sets aside memory for temporarily storing internal data and addresses during subroutine calls and during FOR/NEXT loops. The default is 768 bytes. An "Out of memory" error occurs if there is insufficient stack space for program execution.

**Note:** BASIC allocates string space dynamically. An "Out of string space" error occurs only if no free memory is left for BASIC.

Since CLEAR initializes all variables, you must use it near the beginning of your program, before any variables have been defined and before any DEF statements.

## Examples:

```
CLEAR
```

clears all variables and closes all files.

```
CLEAR, 45000
```

clears all variables and closes all files; makes 45000 the highest address BASIC may use to run your programs.

```
CLEAR, 61000, 200
```

clears all variables and closes all files; makes 61000 the highest address BASIC may use to run your programs, and allocates 200 bytes for stack space.

# CLOSE **Statement**

## CLOSE [*buffer*,. . .]

Closes access to a file.

*Buffer* is a number from 1 – 15 used to OPEN the file. If no buffers are specified, BASIC closes all open files.

This command terminates access to a file through the specified buffer. If a *buffer* was not assigned in a previous OPEN statement, then

```
CLOSE buffer
```

has no effect.

Do not remove a diskette which contains an open file. CLOSE the file first. This is because the last records may not have been written to disk yet. Closing the file writes the data, if it hasn't already been written.

See also OPEN and the chapter on "Disk Files".

## Examples

```
CLOSE 1, 2, 8
```

terminates the file assignments to buffers 1,2, and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST% + COUNT%
```

terminates the file assignment to the buffer specified by the sum FIRST% + COUNT%.

# CLS

Clears the screen.

If the screen is in text mode, CLS clears the active page to the currently selected background color. See COLOR statement. If the screen is in medium or high resolution mode, CLS clears the entire screen buffer to black.

CLS returns the cursor to home position. In graphics mode, home position is the center of the screen. In medium resolution, that is position 160,100. In high resolution, home position is 320,100 or 320,200, depending on the current SCREEN mode.

If a SCREEN or WIDTH statement changes the screen mode, the screen clears for the new mode. You can also clear the screen by pressing (CTRL) and (L) or (CTRL) and (HOME).

## Sample Program

```
540 CLS
550 FOR I = 1 TO 24
560 PRINT STRING$ (79,33)
570 NEXT I
580 GOTO 540
```

# COLOR/Text **Statement**

## COLOR [*foreground, background, border*]

Selects the display colors for the foreground, background, and border on the video display.

*foreground* is an integer in the range 0 to 31, specifying the foreground color.

*background* is an integer in the range 0 to 15, specifying the background color.

*border* is an integer in the range 0 to 15, specifying the border color with the Medium Resolution Color Graphics option. With the High Resolution Color Graphics option the border is always black, and BASIC ignores this parameter.

For more information about the graphics commands, see Chapter 6 "Introduction to Graphics."

The first part of the COLOR/Text description gives the COLOR/Text statement for all computers, regardless of options. An additional description is provided for the color graphics options. Please note the following about the COLOR/Text statement, regardless of any options you are using:

1. To be in text mode, you must have selected Mode 0 with the SCREEN statement.
2. If you omit any parameter, BASIC assumes the previous or the default values.
3. If you set foreground color the same as background color, the characters are invisible.

Possible foregound selections are:

| | |
|---|---|
| 0 or 8 | Black |
| 1 | Underlined white character |
| 2-7 | White |
| 9 | High intensity white underlined |
| 10-15 | High intensity white |
| 16 or 24 | Black blinking |
| 17 | Underlined white blinking |
| 18-23 | White blinking |
| 25 | High intensity white underlined blinking |
| 26-31 | High intensity white blinking |

High intensity white is a brighter white. There is no high intensity black.

Possible background selections are:

| | |
|---|---|
| 0-6 | Black |
| 7 | White |

Specifying white (7) as a background color displays only if the foreground selection is black. The foreground may be 0, 8, 16, or 24. White background with black characters creates a reverse video image.

Specifying black (0-6) as a background color displays only if the foreground selection is white. That is, you may not specify a foreground color selection of 0, 8, 16 or 24.

## Examples

```
COLOR 0,7
```

selects black characters on a white background.

```
COLOR 1, 0
```

selects underlined white characters on a black background.

```
COLOR 4, 0
```

selects white characters on a black background.

# Color Graphics Options

With the Medium or High Resolution Color Graphics options you may select the following colors for foreground and background:

| | |
|---|---|
| 0, 8, 16, or 24 | Black |
| 1 or 17 | Blue |
| 2 or 18 | Green |
| 3 or 19 | Cyan |
| 4 or 20 | Red |
| 5 or 21 | Magenta |
| 6 or 22 | Yellow |
| 7 or 23 | Gray |
| 9 or 25 | Light Blue |
| 10 or 26 | Light Green |
| 11 or 27 | Light Cyan |
| 12 or 28 | Light Red |
| 13 or 29 | Light Magenta |
| 14 or 30 | Light Yellow |
| 15 or 31 | White |

With the Medium Resolution Color Graphics Option, you may also select the border color from the above listing. With the High Resolution Color Graphics Option, the border is always black.

With the Medium Resolution Color Graphics option you may display only five colors at one time. Of the five, one can be the border, one can be the background, and three can be foreground. This means that you can display text in three different colors.

If you execute a COLOR/Text statement that uses a fourth foreground color, the fourth foreground color replaces the first foreground color you selected. All characters of first color change to the fourth color.

You can think of it as a first-in-first-out system. The first foreground color you specify is the first color replaced. The sixth foreground color you select replaces the second color. For example, if you execute the following statements:

```
COLOR 0,6,2:PRINT "PEPPER"
COLOR 7:PRINT "TABBY"
COLOR 4:PRINT "WAYNE"
COLOR 15:PRINT "ROBBIE"
```

The first line prints PEPPER in black on a yellow background with a green border.

The second line prints TABBY in gray. Background and border retain their previous values.

The third line prints the word WAYNE in red. Background and border retain their previous values.

When BASIC executes the fourth line, it requires a fourth foreground color to print ROBBIE in white. White replaces black as one of the three possible foreground colors. BASIC prints ROBBIE in white and also changes PEPPER to white.

With the High Resolution Color Graphics option you can display characters in seven different foreground colors at one time. The principal is the same as with the Medium Resolution Color Graphics option, first-in-first-out. If you select an eighth color, that color replaces the first foreground color. If you select a ninth color, that color replaces the second foreground color.

## Examples

```
COLOR 7,0,0
```

Selects white characters on a black background with a black border.

```
COLOR ,,4
```

Changes border color to red. The foreground and background colors retain their previous values.

```
COLOR 6,1
```

Changes the foreground to yellow and background to blue. Border retains its previous value.

```
COLOR ,6
```

Changes background to yellow. If the previous example has been executed, any characters on the screen are now invisible.

# COLOR/Graphic **Graphic Statement**

## COLOR [*background*] [,*palette*]

Selects the *palette* of colors to be used by subsequent graphics statements.

*background* is an integer in the range 0 to 15 that specifies the background and border colors as described in the COLOR/Text statement. In Screen Modes 2 and 4, the border is always black.

*palette* is a numeric expression in the range 0 to 255 that specifies the palette of colors. Even numbers select *palette* 0, and odd numbers select *palette* 1.

The palette of colors is the group of colors associated with color numbers specified in subsequent graphics statements, such as LINE or PRESET. When you select a palette, you tell BASIC to associate certain colors with position numbers in the palette when you use them as the color parameter in graphics statements.

Color number 0 is the current background color. The other colors and their position numbers when you specify each palette are:

| Position Number | Palette 0 | Palette 1 |
|---|---|---|
| 1 | Green | Cyan |
| 2 | Red | Magenta |
| 3 | Yellow | White |
| 4 | White | Light Red |
| 5 | Light Cyan | Light Green |
| 6 | Light Blue | Light Blue |
| 7 | Light Yellow | Light Yellow |

In Screen Mode 1 you may only specify colors 0, 1, 2, and 3 in your graphics commands.

These colors are the default colors when you execute a COLOR/Graphic statement to select a palette. After executing the COLOR/Graphics statement, you may use the PALETTE and PALETTE USING statements to change any or all of these values. See PALETTE AND PALETTE USING.

Please note the following regardless of which graphics options you are using:

1. To be in color graphics mode, you must have selected Screen Mode 1 or 3 with the SCREEN statement.
2. If you omit any parameters, BASIC assumes the previous values.
3. If you set foreground color the same as background color, the characters are invisible.

## Examples

```
10 COLOR 9,0
```

Sets background to light blue and selects Palette 0.

```
20 COLOR ,3
```

Background retains its previous value. Because 3 is an odd number, Palette 1 is selected.

```
10 COLOR 11,1
20 LINE (0,0) - (319,199),1
```

Line 10 selects a light cyan background and Palette 1. Line 20 draws a cyan diagonal line on the display because the color of Position 1 in Palette 1 is cyan.

```
10 COLOR 3,0
20 LINE (0,0) - (319,199),5
```

Line 10 selects a cyan background and Palette 0. Line 20 draws a light cyan diagonal line on the video display. If you select Palette 1 in Line 10, Line 20 draws a light green diagonal line.

## COM(1) *action*

Turns on, turns off, or temporarily halts the trapping of activity on the communications channel.

*action* may be any of the following:

| | |
|---|---|
| ON | enables communication trapping |
| OFF | disables communication trapping |
| STOP | temporarily suspends communication trapping |

Use the COM statement in a communication trap routine with the ON COM(1) statement to detect when characters have come into the communication channel. The statement

```
COM(1) ON
```

turns the trap on. BASIC checks after every program statement to see if a character has come into the communication channel. If there is activity on the communication channel, BASIC transfers program control to the line number specified in the ON COM(1) statement.

The statement

```
COM(1) STOP
```

temporarily halts communication trapping. If activity occurs on the communication channel, BASIC does not transfer program control to the ON COM(1) statement until communication trapping is turned on again by executing a COM(1) ON statement. BASIC remembers that activity took place. Immediately after communication trapping is turned on again, BASIC transfers program control to the line number specified in the ON COM(1) statement.

The statement

COM(1) OFF

turns off communication activity trapping and does not remember that activity took place when activity trapping is turned on again.

We recommend that your COM trap routine read the entire message from the communication port. Do not use a COM trap to trap for a single character message because the amount of time required to trap and read every character can cause the communication buffer to overflow.

See ON COM(1) for more information about communication trapping.

## Example

```
10 COM(1) ON
20 PRINT "NO ACTIVITY"
30 ON COM(1) GOSUB 100
40 GOTO 20
.
.
100 PRINT "YOU ARE RECEIVING DATA":
.
.
.
200 RETURN
```

Line 10 turns on a communication trap. If characters are received on the communication channel, program control transfers to the subroutine beginning at Line 100. If there is no activity on the communications channel, Line 20 prints a message and Line 40 keeps the program in a loop until there is activity on the communication channel. Note that BASIC checks the communication channel for activity after executing each statement.

# COMMON    Statement

## COMMON *variable,. . .*

Passes *variables* to a CHAINed program.

COMMON may appear anywhere in a program, but we recommend using it at the beginning.

The same variable cannot appear in more than one COMMON statement in a single program. The size and order of the variables must be the same in the programs being CHAINed. To specify array variables, append "( )" to the variable name. If you are passing all variables, use CHAIN with the ALL option and omit the COMMON statement.

**Note:** array variables used in a COMMON statement must have been declared in a DIM statement.

## Example

```
 90 DIM D(50)
100 COMMON A, B, C, D( ),G$
110 CHAIN "PROG3.BAS", 10
```

line 100 passes variables A, B, C, D and G$ to the CHAIN command in line 110.

See also CHAIN.

# CONT — Statement

## CONT

Resumes program execution.

You may only use CONT if the program was stopped by the (BREAK) key, or a STOP or an END statement in the program.

CONT is primarily a debugging tool. During a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT (ENTER); execution continues with the current variable values.

You cannot use CONT after editing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.

## Example

```
10  INPUT A, B, C
20  K=A^2
30  L=B^3/ .26
40  STOP
50  M=C+40*K+100: PRINT M
```

Run this program.
You will be prompted with:

?

Type:

1, 2, 3 (ENTER)

The computer displays:

Break in 40

You can now type any immediate command.

For example:

PRINT L

displays 30.7692. You can also change the value of A, B, or C.

For example:

**C = 4**

changes the value of C in the program. Type:

**CONT**

your screen displays: 144.

See also STOP.

# COS **Function**

## COS (*number*)

Computes the cosine of *number*.

COS returns the cosine of *number* in radians. The *number* must be given in radians. When *number* is in degrees, use COS(*number* * .01745329).

The result is always single precision.

## Examples

```
Y = COS(X * .01745329)
```

stores in Y the cosine of X, if X is an angle in degrees.

```
PRINT COS(5.8) - COS(85 * .42)
```

prints the arithmetic (not trigonometric) difference of the two cosines.

# CSRLIN — Function

## [*variable*] = CSRLIN

Returns the current row position of the cursor.

*variable* is a numeric variable to hold the value returned by CSRLIN. Because there are 24 usable lines on the screen, the value is 1 through 24.

See the POS function to return the current column position and the LOCATE statement to set the row and column positions.

## Sample Program

```
10 PRINT "This is Line ";
20 ROW = CSRLIN
30 PRINT ROW
```

# CSNG                                                    Function

## CSNG (*number*)

Converts *number* to single precision.

If *number* is double precision, when its single-precision value is printed, only six significant digits are shown. BASIC rounds the number in this conversion.

## Example

```
PRINT CSNG(.1453885509)
```

prints .145389

## Sample Program

```
280 V# = 876.2345678#
290 PRINT V#; CSNG(V#)
RUN
 876.2345678       876.2346
Ok
```

# CVD, CVI, CVS — Function

## CVD (*eight-byte string*)
## CVS (*four-byte string*)
## CVI (*two-byte string*)

Convert string values to numeric values.

These functions restore data to numeric form after it is read from disk. Typically, the data has been read by a GET statement, and is stored in a direct access file buffer. CVD converts an *eight-byte string* to a double-precision number. CVS converts a *four-byte string* to a single-precision number. CVI converts a *two-byte string* to an integer.

CVD, CVI, and CVS are the inverses of MKD$, MKI$, and MKS$, respectively.

## Examples

Suppose the name GROSSPAY$ references an eight-byte field in a direct-access file buffer, and after GETting a record, GROSSPAY$ contains an MKD$ representation of the number 13123.38. Then the statement

```
A# = CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

## Sample Program

This program reads from the file "TEST.DAT", which is assumed to have been previously created. For the program that creates the file, see MKD$, MKI$, and MKS$.

```
1420 OPEN "R", 1, "TEST.DAT", 14
1430 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1440 GET 1
1450 PRINT CVI(I1$), CVS(I2$), CVD(I3$)
1460 CLOSE
```

**Note:** GET without a record number tells BASIC to get the first record from the file, or the record following the last record accessed.

# DATA Statement

## DATA *constant,. . .*

Stores numeric and string *constants* to be accessed by a READ statement.

This statement may contain as many *constants* (separated by commas) as will fit on a line. Each will be read sequentially, starting with the first constant in the first DATA statement, and ending with the last item in the last DATA statement.

Numeric expressions are not allowed in a DATA list. If your string constants include leading blanks, colons, or commas, you must enclose these constants in double quotation marks.

DATA statements may appear anywhere it is convenient in a program. The data types in a DATA statement must match up with the variable types in the corresponding READ statement, otherwise a "Syntax error" occurs.

To reREAD DATA statements from the beginning, use a RESTORE statement before the next READ statement.

## Examples

```
1340 DATA NEW YORK, CHICAGO, LOS
     ANGELES, PHILADELPHIA, DETROIT
```

stores five string data items. Note that quote marks aren't needed, since the strings contain no delimiters and the leading blanks are not significant.

```
1350 DATA 2.72, 3.14159, 0.0174533, 57.29578
```

stores four numeric data items.

```
1360 DATA "SMITH, T.H.", 38, "THORN, J.R.", 41
```

stores both types of constants. Quote marks are required around the first and third items because they contain commas (commas are delimiters between constants).

## Sample Program

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

This program READS string and numeric data from the DATA statement in line 30.

# DATE$ — Statement

***variable* = DATE$**
**DATE$ = *string***

Sets or retrieves the current date.

*variable* is a variable in your BASIC program that receives the current date.

*string* is a literal, enclosed in quotes, that sets the current date by assigning a value to DATE$.

## Setting the Date

This system supports dates between January 1, 1980 and December 31, 2099. You may use either a slash or a hyphen to separate the month, day, and year. You may use any of the following forms to set the current date:

| | |
|---|---|
| mm/dd/yy | mm/dd/yyyy |
| mm – dd – yy | mm – dd – yyyy |

mm is the month and may be any number 01 – 12.
dd is the day and may be any number 01 – 31.
yy or yyyy is the year and may be 01 – 99 or 1980 – 2099.

You may omit leading zeroes for the month and day. If you only supply two digits for the year, BASIC precedes these digits with 19.

## Retrieving the Date

Regardless of the form you use to set the date, BASIC does the following when retrieving it:

- Separates month, day, and year with hyphens.
- Displays month and day as two digits, inserting leading zeroes as necessary.
- Displays year in four digits.

## Examples

```
DATE$ = "9/6/83"
```

sets the current date as 09–06–1983.

```
DATE$ = "10/22/83"
```

sets the date as 10–22–1983.

```
DATE$="6/6/86"
```

sets the date as 06–06–1986.

```
CURDATE$= DATE$
```

assigns the value of the current date to the variable CUR-DATE.

# DEFDBL/INT/SNG/STR **Statement**

## DEFDBL *letter,...*
## DEFINT *letter,...*
## DEFSNG *letter,...*
## DEFSTR *letter,...*

Defines any variables beginning with letter(s) as: (DBL) double precision, (INT) integer, (SNG) single precision, or (STR) string.

**Note:** A type declaration tag always takes precedence over a DEF statement.

## Examples

```
10 DEFDBL L-P
```

classifies all variables beginning with the letters L through P as double-precision variables. Their values are stored with 17 digits of precision, though only 16 are printed.

```
10 DEFSTR A
```

classifies all variables beginning with the letter A as string variables.

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

classifies all variables beginning with the letters I through N, W and Z as integer variables. Their values are in the range −32768 to 32767.

```
10 DEFSNG I, Q-T
```

classifies all variables beginning with the letters I or Q through T as single-precision variables. Their values include seven digits of precision, though only six are printed out.

## DEF FN *function name* [ (*argument*,. . .) ] = *function definition*

Defines *function name* according to your *function definition*.

*Function name* must be a valid variable name. This name, preceded by FN, is the name of the function when you call it. The type of variable used determines the type of value the function will return. For example, if you use a single-precision variable, the function will always return single-precision values.

*Argument* represents those variables in *function definition* that are to be replaced when the function is called. If you enter several variables, separate them by commas.

*Function definition* is an expression that performs the operation of the function. A variable used in a function definition may or may not appear in *argument*. If it does, BASIC uses its value to perform the function. Otherwise, it uses the current value of the variable.

Once you define and name a function (by using this statement), you can use it as you would any BASIC function.

## Examples

```
DEF FNR = RND(90)+9
```

defines a function FNR to return a random value between 10 and 99. Notice that the function can be defined with no arguments.

```
210 DEF FNW# (A#,B#)=(A#-B#)*(A#-B#)
280 T = FNW#(I#,J#)
```

defines function FNW# in line 210. Line 280 calls that function and replaces parameters A# and B# with parameters I# and J#. (We assume that I# and J# were assigned values elsewhere in the program).

**Note:** Using a variable as a parameter in a DEF FN statement has no effect on the value of that variable. You may use that variable in another part of the program without interference from DEF FN.

# DEF SEG — Statement

## DEF SEG [= *address*]

Assigns the current segment address.

*address* is an integer in the range 0 to 65535. A value outside this range causes an "Illegal Function Call" error, and BASIC retains the previous value.

If you do not specify *address*, the default value is BASIC's data segment (DS).

If you specify *address*, do so on a 16-byte boundary. BASIC shifts the value left 4 bits and adds the offset specified in the instruction to the value to form the code segment address for the instruction. See the section "Interfacing Assembly Language Subroutines" in Appendix E for more information.

**Note:** BASIC does not check the validity of the resultant segment + offset address.

When you load BASIC, the DS (data segment) register is set to the address of BASIC's workspace. This is the default value of the DS register. You must, therefore, execute a DEF SEG statement before executing BLOAD, BSAVE, PEEK, POKE, USR, or CALL, unless you used the /M: switch when you loaded BASIC. Without the DEF SEG statement or the /M: switch, these statements and functions could destroy BASIC's workspace. If you execute a DEF SEG to change the DS register to a different segment, you must execute another DEF SEG to restore the DS register to its default value.

Separate DEF and SEG with a space. Otherwise, BASIC interprets the statement

```
DEFSEG=100
```

to mean "assign the value 100 to the variable DEFSEG."

## Example

```
10 DEF SEG=&HB800    'Set segment to 800 Hex
20 DEF SEG           'Restore to BASIC data
                      segment
```

# DEF USR **Statement**

## DEF USR[*digit*] = *offset*

Defines the segment offset and user number of a subroutine to be called by the USR function.

*digit* may be an integer in the range 0 to 9.

offset is an integer in the range of 0 to 65535. It specifies the location into a segment where the subroutine begins in memory.

When a USR function is executed, BASIC transfers control to the subroutine through the address given in the last DEF SEG statement and the segment offset specified in the DEF USR statement. If the subroutine is not in BASIC's data segment, a DEF SEG statement must be executed before the USR function. See the section "Interfacing Assembly Language Subroutines" in Appendix E and USR in this chapter for more details.

A program may contain any number of DEF USR statements, allowing access to as many subroutines as necessary. However, only 10 definitions may be in effect at one time.

If you omit *digit*, BASIC assumes USR0.

## Examples

```
DEF USR3 = &H0020
DEF SEG = &H1700
```

USR3 begins at offset hexadecimal 20 in the segment beginning at hexadecimal address 1700. When your program calls USR3, control branches to your subroutine beginning at absolute hexadecimal address 17020.

# DELETE **Statement**

## DELETE *line1 - line2*

Deletes from *line1* through *line2* of a program in memory.

A period (".") can be substituted for either *line1* or *line2* to indicate the current line number.

## Examples

```
DELETE 70
```

deletes line 70 from memory. If there is no line 70, an error will occur.

```
DELETE 50-110
```

deletes lines 50 through 110 inclusive.

```
DELETE -40
```

deletes all program lines up to and including line 40.

```
DELETE -.
```

deletes all program lines up to and including the line that has just been entered or edited.

```
DELETE .
```

deletes the program line that has just been entered or edited.

# DIM — Statement

## DIM *array* (*dimension(s)*), *array* (*dimension(s)*), . . .

Sets aside storage for *arrays* with the *dimensions* you specify.

Arrays may be of any type: string, integer, single precision or double precision, depending on the type of variable used to name the array. If no type is specified, the array is classified as single precision.

When you create the array, BASIC reserves space in memory for each element of the array. All elements in a newly-created array are set to zero (numeric arrays) or the null string (string arrays).

**Note:** The lowest element in a dimension is always zero, unless an OPTION BASE 1 statement is executed.

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a BASIC statement. For example,

```
A(5) = 300
```

creates array A and assigns element A(5) the value of 300. Each dimension of an implicitly-defined array contains 11 elements, subscripts 0-10.

## Examples

```
DIM AR(100)
```

sets up a one-dimensional array AR( ), containing 101 elements: AR(0), AR(1), AR(2), . . . , AR(98), AR(99), and AR(100).

**Note:** The array AR( ) is completely independent of the variables AR.

```
DIM L1%(8,25)
```

sets up a two-dimensional array L1%(,), containing 9 x 26 integer elements, L1%(0,0), L1%(1,0), L1%(2,0), . . . , L1%(8,0), L1%(0,1), L1%(1,1), . . . , L1%(8,1), . . . , L1%(0,25), L1%(1,25), . . . , L1%(8,25).

Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

```
0,0   0,1   0,2   0,3   ...  0,23  0,24  0,25
1,0   1,1   1,2   1,3   ...  1,23  1,24  1,25
.
.
.
7,0   7,1   7,2   7,3   ...  7,23  7,24  7,25
8,0   8,1   8,2   8,3   ...  8,23  8,24  8,25
```

```
DIM B1(2,5,8),   CR(2,5,8),   LY$(50,2)
```

sets up three arrays:
B1(,,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.
LY(,) is two-dimensional, containing 51*3 string elements.

# DRAW **Statement**

## DRAW *direction* [*number*] . . .

Draws an object on the video display.

*direction* specifies one or more of the movement commands listed below.

*number* specifies the number DRAW uses with scale factor to determine the actual distance to move. If you omit *number*, DRAW assumes one. DRAW moves scale factor * *number* points.

# Movement Commands

Each of the following movement commands begin movement from the "current graphics position," which is the coordinate of the last graphics point plotted with another graphics command, such as LINE or PSET. The current position defaults to the center of the screen if no previous graphics command is executed.

| | | |
|---|---|---|
| U | [*number*] | Move up |
| D | [*number*] | Move down |
| L | [*number*] | Move left |
| R | [*number*] | Move right |
| E | [*number*] | Move diagonally up and right |
| F | [*number*] | Move diagonally up and left |
| G | [*number*] | Move diagonally down and left |
| H | [*number*] | Move diagonally down and right |

M *x-coordinate, y-coordinate*

If you precede the coordinates with a plus ( + ) or minus ( – ) sign, DRAW assumes it is a relative position. Otherwise, it is an absolute position.

# Prefix Commands

These prefix commands can precede the movement commands.

| | |
|---|---|
| B | Move but don't plot any points. |
| N | Move but return to original position when done. |
| A*angle* | Set an angle. *angle* may be in the range of 0 to 3. 0 is 0 degrees, 1 is 90 degrees, 2 is 180 degrees, and 3 is 270 degrees. |
| C*color* | Set color number as described in COLOR/ Graphics. *color* may be in the range 0 to 3 in Screen Mode 1, 0 to 7 in Screen Mode 3, and 0 to 1 in Screen Modes 2 and 4. |
| S*integer* | Set scale factor. *integer* may be in the range 1 to 255. The scale factor is *integer* divided by 4. For example, if *integer* is 2, the scale factor is 2/4. To determine the actual travel distance, multiply the scale factor by the *number* in the movement commands. |
| X*variable*; | Executes a substring. The X command allows you to execute a second substring from a string, much like GOSUB. You can have one string execute another, which executes a third, and so on. *variable* is a string variable in your program that contains the substring you want to execute. *variable* may contain an X command to execute another substring. The semicolon after string is required. |

In the prefix commands, the numeric arguments can be constants or variables. If you use a variable name as a numeric argument, you must follow it with a semicolon.

# Sample Programs

```
10 U$ = "U30;": D$= "D30;": L$ = "L40;": R$ =
   "R40;"
20 BOX$ = U$ + R$ + D$ + L$
30 DRAW "XBOX$"
```

draws a rectangle on the screen.

```
10 U$ = "U30;": D$= "D30;": L$ = "L40;": R$ =
   "R40;"
20 DRAW "XU$; XR$; XD$; XL$;"
```

draws the same rectangle as the previous example.

```
10 SCREEN 1
20 DRAW "L40 E20 F20"
```

draws a triangle on the screen.

# EDIT **Statement**

## EDIT *line*

Enters the edit mode so that you can edit *line*.

See the chapter on the "Edit Mode" for more information.

## Examples

```
EDIT 100
```

enters edit mode at line 100.

```
EDIT .
```

enters edit mode at current line.

# END

Ends program execution and closes all files.

This statement may be placed anywhere in the program. It forces execution to end at some point other than the last sequential line.

An END statement at the end of a program is optional.

## Sample Program

```
40 INPUT S1, S2
50 GOSUB 100
55 PRINT H
60 END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

line 60 prevents program control from "crashing" into the subroutine. Line 100 may only be accessed by a branching statement, such as GOSUB in line 50.

# EOF Function

## EOF(*buffer*)

Detects the end of a file.

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid "Input past end" errors during sequential input.

EOF does not accurately detect the end-of-file marker for random files that contain less than 128 bytes. We recommend that you use the LOF function with random access files.

EOF(*buffer*) returns 0 (false) when the EOF record has not been read yet, and −1 (true) when it has been read. The buffer number must access an open file.

## Sample Program

The following sequence of lines reads numeric data from DATA.TXT into the array A(). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop.

```
1470 DIM A(100)        'ASSUMING THIS IS A SAFE
                       VALUE
1480 OPEN "I", 1, "DATA.TXT"
1490 I% = 0
1500 IF EOF(1) THEN GOTO 1540
1510 INPUT#1, A(I%)
1520 I% = I% + 1
1530 GOTO 1500
1540 REM PROG. CONT. HERE AFTER DISK
     INPUT
```

# EOF — Communication Function

## EOF(*variable*)

Detects an empty input queue for communications files.

*variable* is a variable in your BASIC program to receive the value 0 (false) if there are characters in the input queue waiting to be read and −1 (true) if the input queue is empty.

## Sample Program

These lines would be useful in a program when you want to run the program while waiting for communication activity. Line 10 opens a file and allocates Buffer 1 for communication. Line 20 causes BASIC to check for activity on the communications channel after executing every statement. Line 30 instructs BASIC to perform the subroutine beginning at Line 1000 as soon as there is activity on the communication channel. When all of the communication data has been processed, Line 1050 returns to the main program.

```
10 OPEN  "COM1:300, N, 8, 1, ASC" AS 1
20 COM(1) ON
30 ON COM(1) GOSUB 1000
.
.
.
.
1000 'Communication Subroutine Begins Here
.
.
.
1050 IF EOF(3) THEN RETURN
```

# ERASE — Statement

## ERASE *array*, . . .

Erases one or more *arrays* from a program.

This lets you either redimension arrays or use their previously allocated space in memory for other purposes.

If one of the parameters of ERASE is a variable name which is not used in the program, an "Illegal Function Call" occurs.

## Example

```
450 ERASE C,F
460 DIM F(99)
```

line 450 erases arrays C and F. Line 460 redimensions array F.

# ERL

Returns the line number in which an error occurred.

This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number 0 is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, BASIC returns the largest possible line number, 65535.

## Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number in the variable E.

For an example of how to use ERL in a program, see ERROR.

# ERR

Returns the error code.

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO. See Appendix A for a list of Error Codes.

## Example

```
IF ERR = 7 THEN 1000 ELSE 2000
```

branches the program to line 1000 if the error is an "Out of Memory" error (code 7); if it is any other error, control goes instead to line 2000.

For an example of how to use ERR in a program, see ERROR.

# ERROR **Statement**

## ERROR *code*

Simulates a specified error during program execution.

*Code* is an integer expression in the range 0 to 255 specifying one of BASIC's error codes.

This statement is mainly used for testing an ON ERROR GOTO routine. When the computer encounters an ERROR statement, it proceeds as if the error corresponding to that code had occurred. (Refer to Appendix A for a listing of Error Codes and their meanings).

## Example

```
ERROR 1
```

a "Next Without For" error (code 1) "occurs" when BASIC reaches this line.

## Sample Program

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B>5000 THEN ERROR 21 ELSE GOTO 420
400 IF ERR = 21 THEN PRINT "HOUSE LIMIT IS
    $5000"
410 IF ERL = 130 THEN RESUME 500
420 S = S+B
430 GOTO 120
500 PRINT "THE TOTAL AMOUNT OF YOUR BET
    IS";S
510 END
```

This program receives and totals bets until one of them exceeds the house limit.

# EXP Function

## EXP(*number*)

Calculates the natural exponent of *number*.

Returns e (base of natural logarithms) to the power of *number*. This is the inverse of the LOG function; therefore, *number* = EXP(LOG(*number*)). The *number* you supply must be less than or equal to 88.0296.

The result is always single precision.

## Example

```
PRINT EXP(-2)
```

prints the exponential value .1353353

## Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

# FIELD Statement

## FIELD *buffer, length* AS *field name*, . . .

Divides a direct-access *buffer* into one or more fields. Each field is identified by *field name* and is the *length* you specify.

*Field name* must be a string variable.

This divides a direct file buffer so that you can send data from memory to disk and disk to memory. FIELD must be run prior to GET or PUT.

Before "fielding" a buffer, use an OPEN statement to assign that buffer to a particular disk file. You must use the direct access mode, i.e., OPEN "R", . . . . The sum of all field lengths should equal the record length assigned when the file was OPENed.

You may use the FIELD statement any number of times to "re-field" a file buffer. "Fielding" a buffer does not clear the buffers contents; only the means of accessing it. Also, two or more field names can reference the same area of the buffer.

See also the chapter on "Disk Files", OPEN, CLOSE, PUT, GET, LSET, and RSET.

## Example

```
FIELD 3, 50 AS A$, 50 AS B$
```

tells BASIC to assign two 50-byte fields to the variables A$ and B$. If you now print A$ or B$, you will see the contents of the field. Of course, this value would be meaningless unless you have previously used GET to read a 100-byte record from disk.

**Note:** All data — both strings and numbers — must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, and MKD$/CVD) for converting numbers to strings and strings to numbers.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS
ST$, 7 AS ZP$
```

assigns the first 16 bytes of buffer 3 to field NM$; the next 25 bytes to AD$; the next 10 to CY$; the next 2 to ST$; and the next 7 to ZP$.

# FILES **Statement**

## FILES [*filespec*]

Displays the names of the files on a diskette.

If you specify *filespec*, BASIC lists all files that match that file specification. If you specify a drive as part of the *filespec*, then BASIC lists all files that match the specified filename on **that** drive. If you omit *filespec*, FILES lists all files on the current drive.

FILES is similar to the MS-DOS DIR command, except that you can specify which files on which drive you want to list. *Filespec* may contain question marks and asterisks as wild cards. A question mark matches any character in a filename. For example,

```
FILES "PAY???"
```

lists all filenames that begin with the letters PAY followed by any other three or fewer characters.

An asterisk is a short form of several question marks. It matches any characters beginning at that position. For example,

```
FILES "PAY*"
```

lists all files that have PAY as their first three letters.

## Examples

```
FILES
```

lists all files on the current drive

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

lists all files on the current drive with the extension .BAS

```
FILES "PAY?????.BAS"
```

lists all files beginning with PAY followed by any other five or fewer characters, on the current drive, with the extension .BAS

# FIX Function

## FIX(*number*)

Returns the truncated integer of *number*.

All digits to the right of the decimal point are simply chopped off, so the resultant value is a whole number. For a negative, non-whole number X, FIX(X) = INT(X) + 1. For all others, FIX(X) = INT(X).

The result is the same precision as the argument (except for the fractional portion).

## Examples

```
PRINT FIX (2.6)
```

prints 2.

```
PRINT FIX(-2.6)
```

prints −2.

# FOR/NEXT **Statement**

## FOR *variable* = *initial value* TO *final value* [STEP *increment*]

## NEXT [*variable*]

Establishes a program loop.

*variable* must be an integer or single precision numeric constant.

If you omit *increment*, BASIC assumes the value one.

Each FOR/NEXT loop must have a unique variable.

A loop allows for a series of program statements to be executed over and over a specified number of times.

BASIC executes the program lines following the FOR statement until it encounters a NEXT. At this point, it increases *variable* by STEP *increment*. If the value of *variable* is less than or equal to *final value*, BASIC branches back to the line after FOR, and repeats the process. If *variable* is greater than *final value*, it completes the loop and continues with the statement after NEXT.

If *increment* is a negative value, BASIC decreases the variable each time through the loop and the final value is lower than the initial value.

BASIC always sets the final value for the loop variable before setting the initial value.

**Note:** BASIC skips the body of the loop if *initial value* times the sign of STEP *increment* exceeds *final value* times the sign of STEP *increment*.

## Example

```
20 FOR H=1 TO 2 STEP -2
30       PRINT H
40 NEXT H
```

the initial value of H times the sign of STEP increment is greater than the final value of H times the sign of STEP increment, therefore BASIC skips the body of the loop. (The sign of STEP increment is negative in this case.)

## Sample Program

```
820 I=5
830 FOR I = 1 TO I + 5
840       PRINT I;
850 NEXT
RUN
```

this loop is executed ten times. It produces the following output:

```
1  2  3  4  5  6  7  8  9  10
```

## Nested Loops

FOR/NEXT loops may be "nested". That is, a FOR . . . NEXT loop may be placed within the context of another FOR . . . NEXT loop.

The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

## Sample Program

```
880  FOR I = 1 TO 3
890        PRINT "OUTER LOOP"
900              FOR J = 1 TO 2
910                    PRINT TAB(5) "INNER LOOP"
920              NEXT J
930  NEXT I
```

This program performs three "outer loops" and within each, two "inner loops". It produces the following output:

```
OUTER LOOP
         INNER LOOP
         INNER LOOP
OUTER LOOP
         INNER LOOP
         INNER LOOP
OUTER LOOP
         INNER LOOP
         INNER LOOP
```

The NEXT statement can be used to close nested loops by listing the counter variables (but make sure not to type the variables out of order). For example, delete line 920 and change 930 to:

```
NEXT J, I
```

**Note:** In nested loops, if you omit the variable(s) in the NEXT statement, the NEXT statement matches the most recent FOR statement.

# FRE Function

## FRE(*dummy number*) or (*dummy string*)

Returns the number of bytes in memory not being used by BASIC.

## Examples

```
PRINT FRE("44")
```

prints the amount of memory left.

```
PRINT FRE(44)
```

prints the amount of memory left.

# GET **Statement**

## GET *buffer* [,*record*]

Reads a record from a direct-access disk file and places it in a *buffer*.

*record* is an integer in the range 0 to 32767 that specifies which record number you want to access. If you omit *record*, BASIC reads the next sequential record (after the last GET).

Before using GET, you must OPEN the file and assign it a buffer.

When BASIC encounters GET, it reads the record number from the file and places it into the buffer. The actual number of bytes read equals the record length set when the file is OPENed.

## Examples

```
GET 1
```

reads the next record into buffer 1.

```
GET 1, 25
```

reads record 25 into buffer 1.

## GET *buffer, integer*

Transfers data from the communications buffer to the file buffer.

*buffer* must be the same *buffer* assigned to the file in the OPEN statement.

*integer* is the number of bytes to transfer from the communications buffer into the file *buffer*. *integer* cannot exceed the value used in the LEN option of the OPEN COM statement.

**Note:** Because of the low performance associated with telephone line communication, we recommend that you **not** use GET and PUT statements in such applications. Instead, use the other disk I/O statements.

# GET [STEP] (*x-coordinate1, y-coordinate1*) – (*x-coordinate2, y-coordinate2*), *array*

Transfers points from an area on the display to an array.

If you specify the STEP option, the numbers you specify as coordinates are offsets from the most recent graphics point referenced. *x-coordinate* is the number of points in the horizontal direction, and *y-coordinate* is the number of points in the vertical direction. Precede the numbers with a plus ( + ) or minus ( – ) sign to indicate the direction (up, down, left, or right) from the most recent point referenced. The plus sign indicates to add the number (right or up) to the most recent coordinate, and the minus indicates to subtract the number (left or down) from the most recent coordinate.

*x-coordinate1* indicates the x coordinate where the image begins. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4, *x-coordinate* may be in the range 0 to 640.

*y-coordinate1* indicates the y coordinate where the image begins. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4, *y-coordinate* may be in the range 0 to 400. If you omit *x-coordinate1* and *y-coordinate1*, BASIC begins the image at the last point referenced on the screen.

*x-coordinate2* indicates the x coordinate where the image ends at and may be in the same range as *x-coordinate2*.

*y-coordinate2* indicates the y coordinate where the image ends at and may be in the same range as *y-coordinate*.

*array* is an array variable name to hold the image. You may not define the array as a string array, and it must be dimensioned large enough to hold the entire image.

You use the GET/Graphics and PUT/Graphics statements together for animation and high-speed object motion in Screen Modes 1, 2, 3, and 4. The GET/Graphics statement transfers the screen image described by specified points of the rectangle into the array. The PUT/Graphics statement transfers the image from the array to the display.

The coordinates that you specify are opposite corners of the image to store in the array. The array is used as a place to hold the image. It may be any numeric precision. To ensure that array is large enough to hold the image use the following formula:

4 + (INT((*h* * *bits per point* + 7)/8) * *v*)

*bits per point* is 2 in Screen Mode 1, 1 in Screen Modes 2 and 4, and 3 in Screen Mode 3. *h* is the length of the horizontal side of the image, and *v* is the length of the vertical side of the image. The dimensions of the image are in points.

If you want to use the GET/Graphics statement to store an image that is 10 by 12 in Screen Mode 1, your array has to be 40 bytes. Determine this by substituting values in the formula as follows:

4 + (INT((10 * 2 +7)/8) * 12) = 40

The array must store 40 bytes. The number of bytes per element of an array are:

2 for integer
4 for single precision
6 for double precision

For this example, you need an integer array with 20 elements, or a single-precision array with 10 elements, or a double-precision array with 7 elements.

If you use an integer array, you can examine the array. Remember, the GET/Graphics statement stores the data in bits. The information from the display is stored in the array as:

| | |
|---|---|
| Element 0 | the x dimension of the image |
| Element 1 | the y dimension of the image |

The remaining elements of the array store the data bits of the image. Numeric data is stored low byte first and then high byte, but the data is transferred high byte first and then low byte.

For more information on using the GET/Graphics and PUT/Graphics statements for high speed animation, see the PUT/Graphics statement.

# GOSUB **Statement**

## GOSUB *line*

Branches to a subroutine, beginning at *line*.

You can call subroutine as many times as you want. When the computer encounters RETURN in the subroutine, it returns control to the statement which follows GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement. Every subroutine must end with a RETURN.

## Example

```
GOSUB 1000
```

branches control to the subroutine beginning at line 1000.

## Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

transfers control from line 260 to the subroutine beginning at line 280. Line 290 instructs the computer to return to the statement immediately following GOSUB.

# GOTO Statement

## GOTO *line*

Branches to the specified *line*.

When used alone, GOTO *line* results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. This lets you pass values assigned in the command mode to variables in the execute mode.

## Example

```
GOTO 100
```

automatically transfers control to line 100.

## Sample Program

```
10 READ R
20 IF R = 13 THEN GOTO 80
30 PRINT "R=";R
40 A=3.14*R^2
50 PRINT "AREA =";A
60 GOTO 10
70 DATA 5,7,12, 13
80 END
RUN
```

line 10 reads each of the data items in line 70; line 60 returns program control to line 10. This enables BASIC to calculate the area for each of the data items, until it reaches item 13.

# HEX$ Function

## HEX$(*number*)

Calculates the hexadecimal value of *number*.

HEX$ returns a string representing the hexadecimal value of the argument. The value returned is like any other string: it cannot be used in a numeric expression.That is, you cannot add hex strings. You can concatenate them, though.

## Examples

```
PRINT HEX$(30), HEX$(50), HEX$(90)
```

prints the following strings:

```
1E      32      5A
```

```
Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

# IF . . . THEN . . . ELSE Statement

## IF *expression* THEN *statement(s)* or *line* [ELSE *statement(s)* or *line*]

Tests a conditional expression and makes a decision regarding program flow.

If *expression* is true, control proceeds to the THEN *statement* or *line*. If not, control jumps to the matching ELSE *statement, line*, or to the next program line.

## Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" : END
```

passes control to PRINT, then to END if X is greater than 127. If X is not greater than 127, control jumps down to the next line in the program, skipping the PRINT and END statements.

```
If A < B THEN PRINT "A < B" ELSE PRINT "B < A"
```

tests the first expression, if true, prints A < B . Otherwise, the program jumps to the ELSE statement and prints B < A.

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

assigns the value X + 180 to Y if both expressions are true. Otherwise, control passes directly to the next program line, skipping the THEN clause.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN
400 ELSE 370
```

branches to line 210 if A$ is YES. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO, then the program branches to line 400. If A$ is any value besides NO or YES, the program branches to line 370.

## Sample Program

IF/THEN/ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs. (IF the statement does not contain the same number of ELSE's and IF's, each ELSE is matched with the closest unmatched IF.)

```
1040 INPUT "ENTER TWO NUMBERS"; A, B
1050 IF A <= B THEN IF A < B THEN PRINT A; ELSE
       PRINT "NEITHER"; ELSE PRINT B;
1060 PRINT "IS SMALLER THAN THE OTHER"
```

This program prints the relationship between the two numbers entered.

# INKEY$

Returns a keyboard character.

Returns a one-character string from the keyboard without having to press (ENTER). If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the display.

INKEY$ is invariably put inside some sort of loop. Otherwise a program execution would pass through the line containing INKEY$ before a key could be pressed.

## Example

```
10 A$ = INKEY$
20 IF A$ = " " THEN 10
```

This causes the program to wait for a key to be pressed.

# INP **Function**

## INP(*port*)

Returns the byte read from a *port*.

INP is the complementary function of the OUT statement.

*Port* may be any integer from 0 to 65535.

## Example

```
100 A=INP(255)
```

# INPUT **Statement**

## INPUT[;] [*"prompt string"*;] *variable1, variable2, . . .*

Inputs data from the keyboard into one or more *variables*.

When BASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type data.

INPUT may specify a list of string or numeric variables, indicating string or numeric data items to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, in that order.

The number of data items you supply must be the same as the number of variables specified. You must separate data items by commas.

Responding to INPUT with too many items, or with the wrong type of value (including numeric type), causes BASIC to print the message "?Redo from start". No values are assigned until you provide an acceptable response.

If a *prompt string* is included, BASIC prints it, followed by a question mark. This helps the person inputting the data to enter it correctly. If instead of a semicolon, you type a comma after *prompt string*, BASIC suppresses the question mark when printing the prompt. *Prompt string* must be enclosed in quotes. It must be typed immediately after INPUT.

If INPUT is immediately followed by a semicolon, any carriage returns pressed as part of the response are not echoed.

## Examples

```
INPUT Y%
```

when BASIC reaches this line, you must type any number and press (ENTER) before the program will continue.

```
INPUT SENTENCE$
```

when BASIC reaches this line, you must type in a string. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME,
AGE)"; N$, A
```

prints a message on the screen to help the person at the keyboard enter the right kind of data.

## Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT"
   CINT(X * .38) "POUNDS."
```

# INPUT# Statement

## INPUT# *buffer, variable, . . .*

Inputs data from a sequential disk file and stores it in a program *variable.*

*Buffer* is the number used when the file was OPENed for input.

*Variable* contains the variable name(s) that will be assigned to the item(s) in the file.

With INPUT#, data is input sequentially. That is, when the file is OPENed, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and re-OPEN it.

INPUT# doesn't care how the data was placed on the disk — whether a single PRINT# statement put it there, or whether it required ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Numeric values: BASIC begins input at the first character which is neither a space or a carriage return. It ends input when it encounters a space, carriage return, or a comma.

String values: BASIC begins input with the first character which is neither a space nor carriage return. It ends input when it encounters a carriage return or comma. One exception to this rule: If the first character is a quotation mark ("), the string will consist of all characters between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character.

If the end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

## Examples

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

sequentially inputs two numeric data items from disk and places them in A and B. Buffer #1 is used.

```
INPUT#4, A$, B$, C$
```

sequentially inputs three string data items from disk and places them in A$, B$, and C$. Buffer #4 is used.

# INPUT$ — Statement

## INPUT$(*number1* [,*number2*])

Inputs a string of characters from either the keyboard or a sequential disk file.

*Number1* is the number of characters to be input. It must be a value in the range 1 to 255. *Number2* is a buffer which accesses a sequential input file.

INPUT$(*number1*) inputs a string of characters from the keyboard. When the program reaches this line, it stops until you (or any operator) type *number1* characters. (You don't need to press (ENTER) to signify end-of-line.) The character(s) you type are not displayed on the screen. Any character, except (BREAK), is accepted for input.

INPUT$(*number1, number2*) inputs a string from a sequential disk file. *Number 2* is the buffer associated with that disk file.

## Examples

```
A$ = INPUT$(5)
```

assigns a string of five keyboard characters to A$. Program execution is halted until the operator types five characters.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A$. The characters are read from the disk file associated with buffer 3.

## Sample Programs

This program shows how you could use INPUT$ to have an operator input a password for accessing a protected file. By using INPUT$, the operator can type in the password without anyone seeing it on the video display. (To see the full file specification, run the program, then type PRINT F$).

```
110 LINE INPUT "TYPE IN THE FILESPEC.EXT"; F$
120 PRINT "TYPE IN THE PASSWORD — MUST
    TYPE 8 CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ + "." + P$
```

In the program below, line 100 OPENs a sequential input file (which we assume has been previously created). Line 200 retrieves a string of 70 characters from the file and stores them in T$. Line 300 CLOSEs the file.

```
100 OPEN "I", 2, "TEST.DAT"
200 T$ = INPUT$(70,2)
300 CLOSE
```

# INSTR **Function**

## INSTR([*integer,*] *string1, string2*)

Searches for the first occurrence of *string2* in *string1,* and returns the position at which the match is found.

*Integer* specifies a position in *string1* to begin searching for *string2. integer* must be a value in the range 1 to 255. If you omit *integer*, INSTR starts searching at the first character in *string1*.

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, it returns zero. Note that the entire substring must be contained in the search string, or zero is returned.

Optional *integer* sets the position for starting the search. If omitted, INSTR starts searching at the first character in *string1*.

## Examples

In these examples, A$ = "LINCOLN":

```
INSTR(A$, "INC")
```

returns a value of 2.

```
INSTR(A$, "12")
```

returns a zero.

```
INSTR(A$, "LINCOLNABRAHAM")
```

returns a zero. For a slightly different use of INSTR, look at

```
INSTR (3, "1232123", "12")
```

which returns 5.

## Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761—) and returns that number. The zip code is

preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
390 READ ADDRESS$
395 IF ADDRESS$ = "$END" THEN 410
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN
    COUNTER = COUNTER + 1 ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX
    ADDRESSES IS" COUNTER: END
420 DATA "5950 GORHAM DRIVE, BURLESON,
    TX *76148"
430 DATA "71 FIRSTFIELD ROAD,
    GAITHERSBURG, MD *20760"
440 DATA "1000 TWO TANDY CENTER, FORT
    WORTH, TX *76102"
450 DATA "16633 SOUTH CENTRAL
    EXPRESSWAY, RICHARDSON, TX *75080"
460 DATA "$END"
```

# INT **Function**

## INT(*number*)

Converts *number* to integer value.

This function returns the largest integer which is not greater than the *number*.

The result has the same precision as the argument except for the fractional portion. *Number* is not limited to the range — 32768 to 32767.

## Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT (-12.11)
```

prints −13.

# KEY/Set/Display **Statement**

## KEY *integer, string*
## KEY ON
## KEY OFF
## KEY LIST

# KEY *integer, string*

Assigns or displays function key values.

*integer* is a number 1 through 12 that indicates the function key being defined.

*string* is the string expression assigned to the key and may contain up to 15 characters.

A soft key is a function key that is "programmed" to generate a specific string of characters. When you press the key, BASIC displays the string on the screen just as if you had typed every character. Initially, the function keys have these soft key values:

| | |
|---|---|
| F1 LIST | F7 TRON(ENTER) |
| F2 RUN(ENTER) | F8 TROFF(ENTER) |
| F3 LOAD" | F9 KEY |
| F4 SAVE" | F10 SCREEN 0,0,0(ENTER) |
| F5 CONT(ENTER) | F11 (none) |
| F6 ,"LPT1:"(ENTER) | F12 (none) |

Functions Keys 11 and 12 do not have initial values. You can use the KEY statement to define these keys. You can also use the KEY statement to redefine the other function keys so that BASIC displays the strings you use most often.

Assigning a string length of zero ("") to a function key disables it as a soft key. For example,

```
KEY 1, " "
```

removes the present capability of the F1 key.

# KEY ON

KEY ON displays the function key assignment values on Line 25 of the screen. If the screen width is 40, the screen shows 5 of the soft key assignments. If the width is 80, the

screen shows 10 of the key assignments. In both cases the screen shows only the first 6 characters of the *string* assignment. When you load BASIC, KEY ON is the initial default value.

(CTRL) (T) has the same effect as a KEY ON statement. If the screen width is 40, KEY ON displays 5 of the soft key assignments. If you press (CTRL) (T), the next five key assignments are displayed. Pressing (CTRL) (T) a second time displays the assignments for Function Keys 11 and 12. This is also true for width 80. KEY ON displays the Key assignments for Function Keys 1 through 10. Pressing (CTRL) (T) displays the assignments for Function Keys 11 and 12.

# KEY OFF

KEY OFF erases the soft key assignments from line 25. The assignments are still active, but the screen does not display them.

BASIC reserves line 25 for soft key display. Even if the soft key display is turned off, BASIC does not display program lines on line 25.

# KEY LIST

KEY LIST displays all 15 characters of all 12 soft key assignments on the screen.

# REMARKS

If a function key has been pressed, an INKEY$ statement in a BASIC program returns one character of a soft key assignment each time it is executed. For example, if this statement is executed

```
A$ = INKEY$
```

and you press F1, the first time the statement is executed, A$ equals L, the second time A$ equals I, and so on. Keep this in mind when writing a BASIC routine to trap for a certain key. Your routine may not perform as expected if a function key is accidentally pressed.

# KEY/Trap **Statement**

## KEY (*number*) *action*

Turns on, turns off, or temporarily halts key trapping for a specified function key or cursor direction key.

*action* may be any of the following:

| | |
|---|---|
| ON | enables key trapping |
| OFF | disables key trapping |
| STOP | temporarily suspends key trapping |

*number* may be a number in the range 1 to 16, indicating the number of the key to trap. Function keys use their corresponding function key number. The cursor direction key trap numbers are

| | |
|---|---|
| ↑ | 13 |
| ← | 14 |
| → | 15 |
| ↓ | 16 |

**Note: Do not** confuse the KEY/Trap statement with the KEY/Display/Set statement. These are two separate statements that perform two distinct functions in BASIC.

The KEY/trap statement is used in a key trapping routine with the ON KEY( ) GOSUB statement to detect when a specific function or cursor direction key is pressed. After executing a KEY( ) ON statement, BASIC checks after each program statement to see if the specified key has been pressed. If so, BASIC transfers program control to the line number specified in the ON KEY( ) GOSUB statement. For example, the statements

```
KEY(3) ON
ON KEY(3) GOSUB 1000
```

turn on a trap for Function Key 3. BASIC continues to execute the other program statements, checking after each statement to see if Function Key 3 has been pressed. When Function Key 3 is pressed, program control branches to the subroutine beginning at Line 1000.

The statement

```
KEY( ) STOP
```

temporarily halts trapping for the specified key. If the key is pressed, BASIC does not transfer program control to the subroutine until key trapping is turned on again with a KEY( ) ON statement. BASIC remembers that the key was pressed and transfers program control to the subroutine immediately after key trapping is turned on again.

The statement

```
KEY( ) OFF
```

turns off key trapping and does not remember that the key was pressed when key trapping is turned on again.

Key trapping only occurs when BASIC is in execution mode. The function keys retain their soft key values during command mode.

See ON KEY( ) GOSUB for more information on key trapping.

## Example

```
10 KEY(1) ON
20 KEY(3) ON
30 KEY(3) STOP
40 KEY(2) OFF
50 ON KEY(1) GOSUB 1000
60 ON KEY(3) GOSUB 2000
.
.
1000 SUBROUTINE
.
.
1100 KEY(3) ON
1110 RETURN
```

Lines 10 and 20 turn on key trapping for Function Keys 1 and 3. Line 30 temporarily suspends key trapping for Function Key 3, and Line 40 turns key trapping off for Function Key 2. This is useful if you want to trap for certain keys to be pressed in a specific sequence. In this example, if Function Key 3 is pressed before Function Key 1, the subroutine for Function Key 3 is not executed until the end of Function Key 1 subroutine, Line 1100. When BASIC executes Line 1110, if Function Key 3 has been pressed, the subroutine beginning at Line 2000 is executed.

# KILL Statement

## KILL *filespec*

"Kills" (deletes) *filespec* from disk.

You may KILL any type of disk file. However, if the file is currently OPEN, a "File already open" error occurs. You must CLOSE the file before deleting it.

## Example

```
KILL "FILE.BAS"
```

deletes this file from the first drive which contains it.

```
KILL "A:DATA"
```

deletes this file from Drive A: only.

# LEFT$ **Function**

## LEFT$(*string,integer*)

Returns the leftmost *integer* characters of *string*.

*integer* must be in the range of 1 to 255.

If *integer* is equal to or greater than LEN (string), the entire string is returned.

## Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

since BIG FIERCE DOG is less than 20 characters long, the whole phrase is printed.

## Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When this is run, BASIC prints:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

Line 750 gets the three leftmost characters of A$ and stores them in B$. Line 760 prints these three characters, a string, and the original contents of A$.

# LEN **Function**

## LEN(*string*)

Returns the number of characters in *string*. Blanks are counted.

## Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

```
PRINT LEN("WAUKEGAN, ILLINOIS")
```

prints 18

# LET **Statement**

## [LET] *variable* = *expression*

Assigns the value of *expression* to *variable*.

BASIC doesn't require assignment statements to begin with LET, but you might want to use LET to be compatible with versions of BASIC that do require it.

## Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X = X - Z1
```

In each case, the variable on the left side of the equal sign is assigned the value of the constant or expression on the right side.

## Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P = "P
```

## LINE [STEP] [(*x-coordinate1, y-coordinate1*)] – (*x-coordinate2, y-coordinate2*) [,*color*] [,B[F]]

Draws a line or a box on the video display.

*x-coordinate1* indicates the x coordinate at which to begin the line. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4, *x-coordinate* may be in the range 0 to 640.

*y-coordinate1* indicates the y coordinate at which to begin the line. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4, *y-coordinate* may be in the range 0 to 400. If you omit *x-coordinate1* and *y-coordinate1*, BASIC begins the line at the last point referenced on the screen.

*x-coordinate2* indicates the x coordinate at which to end the line at and may be in the same range as *x-coordinate1*.

*y-coordinate2* indicates the y coordinate at which to end the line at and may be in the same range as *y-coordinate1*.

If you include the STEP option, the numbers you specify as coordinates are offsets from the most recent graphics point referenced. *x-coordinate* is the number of points in the horizontal direction and *y-coordinate* is the number of points in the vertical direction. Precede the numbers with a plus ( + ) or minus ( – ) sign to indicate the direction (up, down, left, or right) from the most recent point referenced. The plus sign indicates to add the number to the most recent coordinate (right or up) and the minus indicates to subtract the number (left or down) from the most recent coordinate.

*color* indicates the color of the line and must be a color number in the current palette. In Screen Mode 1, *color* may be in the range 0 to 3. In Screen Mode 3, *color* may be in the range 0 to 7. In Screen Modes 2 and 4, *color* may be either 0 or 1. If you omit *color* in Screen Modes 1 or 3, BASIC assumes color 3. If you omit *color* in Screen Modes 2 or 4, BASIC assumes white.

With the **B** option, BASIC draws a box. The points that you specify are opposite corners.

If you specify both the **B** and **F** options, BASIC draws a box and fills the box in with *color*.

If you specify coordinates that are not within the range for the selected screen mode, BASIC assumes the closest legal value. In other words, negative values become zero. In Screen Modes 1 and 2, y values greater than 199 become 199. In Screen Mode 1, x values greater than 319 become 319. In Screen Modes 2, 3, and 4, x values greater than 639 become 639.

## Examples

You can try these examples in Screen Modes 1, 2, 3, or 4. The color, size, and position of the image on the display varies, depending on the current screen mode.

```
LINE -(319, 199)
```

draws a line from the last point referenced to point 319,199 in the default color. This is the simplest form of the LINE statement. Note that when you omit the beginning points you must still include the hyphen.

```
LINE (0,0)-(319,199)
```

draws a diagonal line on the display in the default color.

```
LINE (0,100)-(319,100),1
```

draws a vertical line across the display in Color 1.

```
LINE (0,0)-(320,100),,B
```

draws a box in the upper left corner of the display.

```
LINE (0,0)-(200,200),1,bf
```

draws a box on the display and fills it in with Color 1.

## Sample Programs

```
10 CLS
20 LINE -(rnd*319,rnd*199),rnd*4
30 GO TO 20
```

In Screen Modes 1, 2, 3, or 4, Lines 10-30 create a loop that draws random lines on the video display.

```
40 FOR x=0 TO 319
50 LINE (x,0)-(x,199),x AND 1
60 NEXT
```

In Screen Modes 1, 2, 3, or 4, Lines 40-60 draw an alternating pattern, turning the line on and off.

```
10 CLS
20 LINE -(rnd*639,rnd*199),rnd*2,bf
30 GO TO 20
```

This program draws a random filled box in Screen Modes 2, 3, or 4.

# LINE INPUT — Statement

## LINE INPUT[;][*"prompt message";*] *string variable*

Inputs an entire line (up to 254 characters) from the keyboard.

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, etc.).

LINE INPUT (the space is *not* optional) is similar to INPUT, except:

- The computer does not display a question mark when waiting for input.
- Each LINE INPUT statement can assign a value to only one variable.
- Commas and quotes can be used as part of the string input.
- Leading blanks are not ignored — they become part of variable.

The only way to terminate the string input is to press (ENTER). However, if LINE INPUT is immediately followed by a semicolon, pressing (ENTER) does not echo a carriage return to the display.

Some situations require that you input commas, quotes, and leading blanks as part of the data. LINE INPUT serves well in such cases.

## Examples:

```
LINE INPUT A$
```

inputs A$ without displaying any prompt.

```
LINE INPUT "LAST NAME, FIRST NAME? "; N$
```

displays a prompt message and inputs data. Commas do not terminate the input string, as they do in an INPUT statement.

You may abort a LINE INPUT statement by pressing (BREAK). BASIC returns to command level and displays Ok. Typing CONT resumes execution at LINE INPUT.

# LINE INPUT# **Statement**

## LINE INPUT# *buffer, variable*

Inputs an entire line of data from a sequential disk file to a string *variable*.

*Buffer* is the number under which the file was OPENed.

This statement is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

- the end-of-file
- the 255th data character
- a carriage return

Other characters encountered — quotes, commas, leading blanks — are included in the string.

## Example

If the data on disk looks like this:

```
10 CLEAR 500
20 OPEN "I", 1, "PROG"
```

then the statement

```
LINE INPUT#1, A$
```

could be used repetitively to read each program line, one at a time.

# LIST **Statement**

## LIST [*startline*]-[*endline*] [,*device*]

Lists a program in memory to the display.

*Startline* specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

*Endline* specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program. If you omit *startline* and *endline*, BASIC lists the entire program.

*Device* may be either "SCRN:" (screen) or "LPT1:" (line printer 1). If you omit *device*, the lines are listed to the screen.

You can substitute period (.) for either *startline* or *endline* to signify current line number.

## Examples

```
LIST
```

displays the entire program. If you omit device, you can stop the automatic scrolling by pressing (BREAK). This freezes the display. Press any key to continue the listing. Listings directed to a device may not be interrupted.

```
LIST 50
```

displays line 50 on the screen.

```
LIST 50-85, "SCRN:"
```

displays lines in the range 50-85 on the screen.

```
LIST .-
```

displays the program line that has just been entered or edited, and all higher-numbered lines on the screen.

```
LIST -227
```

displays all lines up to and including 227 on the screen.

```
LIST 227-, "LPT1:"
```

lists line 227 and all higher numbered lines to the printer.

# LLIST **Statement**

## LLIST [*startline*]-[*endline*]

Lists program lines in memory to the printer.

*Startline* specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

*Endline* specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program. If you omit *startline* and *endline*, BASIC lists the entire program.

LLIST assumes a 132-character-wide printer. You may change this by using the WIDTH statement.

## Examples

```
LLIST
```

lists the entire program to the printer. To stop this process, press (HOLD). This causes a temporary halt in the computer's output to the printer. Press any key to continue printing.

```
LLIST 68-90
```

prints lines in the range 68-90.

# LOAD — Statement

## LOAD *filespec* [,R]

Loads a BASIC program into memory.

*filespec* is a string expression containing the drive identifier and filename. The filename is required. If you omit the drive identifier, BASIC assumes the current drive.

If the filename is 8 characters or fewer and you do not specify an extension, BASIC appends the extension **.BAS**.

**Note:** You can press (BREAK) at any time during LOAD, between files, or after a time-out period. BASIC exits the search and returns to direct mode. Previous memory contents remain unchanged.

The R option tells BASIC to run the program. (LOAD with the R option is equivalent to the command RUN *filespec*).

LOAD without the R option wipes out any resident BASIC program, clears all variables, and CLOSES all OPEN files. LOAD with the R option leaves all OPEN files open and runs the program automatically.

You can use either of these commands inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-BASIC file, a "Direct statement in file" error occurs.

## Example

```
LOAD "A:PROG1.BAS"
```

loads PROG1.BAS from Drive A. BASIC then returns to the command mode.

```
LOAD "PROG1.BAS"
```

loads PROG1.BAS since no drive is specified, BASIC begins searching for it in the MS-DOS default drive.

# LOC **Function**

## LOC(*buffer*)

Returns the current record number.

*Buffer* is the buffer under which the file was OPENed.

You use LOC to determine the current record number, that is, the number of the last record processed since the file was OPENed. It returns the record number accessed by the last GET or PUT statement.

## Example

```
IF LOC(1)>55 THEN END
```

if the current record number is greater than 55, ends program execution.

## Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD"
      LOC(1): CLOSE: END
1340 GOTO 1320
```

This is a **portion** of a program. Elsewhere the file has been OPENed and FIELDed. N$ is a field variable. If N$ matches A$, the record number in which it was found is printed.

# LOC Communication Function

## LOC(*variable*)

Returns the number of characters in the input queue.

*variable* is a variable in your BASIC program to receive the number of characters in the input queue waiting to be read.

The input queue can hold more than 255 characters. You determine the number of characters to be stored in the input queue by the value of the /C: switch when BASIC is loaded. Since a string is limited to 255 characters, this eliminates the need for testing string size before reading data into the input queue.

If more than 255 characters are in the input queue, LOC always returns 255. If there are less, LOC returns the actual number of characters waiting to be read.

## Example

```
10 LOC(X)
20 If X>0 THEN 1000
```

Line 10 checks to see if there are any characters in the input queue and stores the number of characters in the variable X. Line 20 tests the value of X. If X is greater than 0, there are characters in the input queue and line 20 transfers program control to line 1000 to process the data.

# LOCATE Statement

## LOCATE [*row*] [,*column*] [,*cursor*] [,*start*] [,*stop*]

Positions the cursor on the screen.

*row* is a numeric expression in the range 1 to 24 that indicates the screen row on which you want to position the cursor. Note that line 25 is reserved for function key values only. You may not use LOCATE to position the cursor on the 25th line.

*column* is a numeric expression that indicates the screen column on which you want to position the cursor. It may be in the range 1 to 40 or 1 to 80, depending on the current screen width.

*cursor* indicates whether the cursor is visible or invisible. Set *cursor* to 1 for a visible cursor and to 0 for an invisible cursor.

*start* is a numeric expression in the range 0 to 7 that specifies the size of the cursor. Values 0, 1, 2, and 3 indicate a full cursor. Values 4, 5, 6, and 7 indicate a half cursor.

The *stop* parameter has no effect in this implementation of BASIC. However, values supplied for *stop* are accepted and ignored to provide compatibility with other implementations of BASIC that use *stop*.

## Examples

```
LOCATE 10,20,1,4
```

positions a half visible cursor on row 10 in column 20.

```
LOCATE 24,1,1,3
```

positions a full cursor in the first position of the last line.

# LOF Function

## LOF(*buffer*)

Returns the length of the file in bytes.

*buffer* is an integer in the range 1 to 15. It is the I/O buffer you used to OPEN the file.

If BASIC creates the file, LOF always returns the number of bytes in the file as a multiple of 128. For example, if the file actually contains 300 bytes, LOF returns 384. If you create the file with EDLIN, LOF returns the actual number of bytes used.

## Example

```
Y = LOF(5)
```

assigns the length of the file in bytes to variable Y.

## Sample Programs

During direct access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

```
1540 OPEN "R", 1, "UNKNOWN.TXT", 128
1550 FIELD 1, 255 AS A$
1560 RECNUM% = 1      'START AT BEGINNING
                       OF FILE
1570 RECSIZE% = 128   'SET RECORD SIZE
1580 IF RECNUM% * RECSIZE% > LOF(1) GOTO
     1640
1590                  'CHECK FOR END OF
                       FILE
1600 GET 1, RECNUM%   'RECORD NUM. TO BE
                       ACCESSED
1610 PRINT A$
1620 RECNUM% = RECNUM% + 1
                      'INCREMENT RECORD
                       NUM
1630 GOTO 1580
1640 CLOSE
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an error.

When you want to add to the end of a file, LOF tells you where to start adding:

```
1700 RECNUM% = (LOF(1) / RECSIZE%) +1
1710                          'HIGHEST EXISTING RE-
                              CORD
1610 PUT 1, RECNUM%  'ADD NEXT RECORD
```

# LOF Communication Function

## LOF(*variable*)

Returns the amount of free space in the input queue.

*variable* is a variable in your BASIC program that receives the amount of free space in the input queue.

You can use LOF to determine when an input queue is getting full so that transmission is stopped.

# LOG **Function**

## LOG(*number*)

Computes the natural logarithm of *number*.

*Number* must be greater than zero. This is the inverse of the EXP function. The result is always in single precision.

## Examples

```
PRINT LOG(3.14159)
```

prints the value 1.14473.

```
Z = 10 * LOG(Ps/P1)
```

performs the indicated calculation and assigns the value to Z.

## Sample Program

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL
    (MILES)"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE
    SPACE IS" L "DECIBELS."
```

# LPOS Function

## LPOS(*number*)

Returns the logical position of the print head within the line printer's buffer.

*Number* indicates which printer and may be any of the following :

| | |
|---|---|
| 0 or 1 | indicates LPT1: |
| 2 | indicates LPT2: |

LPOS is only useful to check the position of the print head after printing an LPRINT statement that is terminated by a semicolon to suppress the automatic carriage return. The statement containing LPOS is not executed until the LPRINT statement is finished printing.

## Examples

```
LPRINT A; B; C;
```

You may want to use LPOS to determine if there is enough room to continue printing more variables on the same line.

```
100 IF LPOS(X)>60 THEN LPRINT
```

If the printer has printed more than 60 characters, a carriage return is sent so that the printer skips to the next line.

LPOS does not necessarily give the physical position of the print head if the printed string contains the ASCII code for a carriage return. For example, if you are printing a string of 20 characters and the 10th character is the ASCII code for a carriage return, after printing the ninth character, the printer advances to the next line and prints the remaining 10 characters. If the string is terminated by a semicolon to suppress the automatic line feed, the physical location of the print head is at position 10, but LPOS returns a value of 21 because that is the logical location of the print head.

# LPRINT, LPRINT USING **Statement**

## LPRINT *data, . . .*
## LPRINT USING *format; data, . . .*

Prints *data* on the printer.

LPRINT and LPRINT USING assume a 132-character-wide-printer. You may change the width with the WIDTH statement.

See PRINT and PRINT USING for more information.

## Examples

```
LPRINT (A * 2)/3
```

prints the value of expression (A * 2)/3 on the printer.

```
LPRINT TAB(50) "TABBED 50"
```

moves the line printer carriage to TAB position 50 and prints "TABBED 50". (Refer to the TAB function).

```
LPRINT USING "#####.#"; 2.17
```

sends the formatted value bbbb2.2 to the line printer.

# LSET Statement

## LSET *field name* = *data*

Sets *data* in a direct-access buffer *field name* in preparation for a PUT statement.

You must have used FIELD to set up buffer fields before using LSET.

You must convert numeric values to string values before they are LSET. See MKI$, MKD$, MKS$.

You use LSET to left-justify the variable in the field. If the field is larger than the variable it is receiving, the field is filled with blanks on the right. If the variable is larger than the field, characters are truncated on the right. The complement command to LSET is RSET.

See also the chapter on "Disk Files", OPEN, CLOSE, FIELD, GET, PUT, and RSET.

## Example

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

```
LSET NM$ = "JIM CRICKET, JR."
LSET AD$ = "2000 EAST PECAN ST."
```

set the data in the buffer as follows:

JIMƀCRICKET,JR.ƀƀƀ 2000ƀEASTƀPECANƀST.ƀƀƀƀƀƀ

Notice that filler blanks are placed to the right of the data strings in both cases. If we use RSET statements instead of LSET, the filler spaces are placed to the left. This is the only difference between LSET and RSET.

# MERGE Statement

## MERGE *filespec*

Loads a BASIC program and merges it with the program currently in memory.

*Filespec* is a string expression, enclosed in quotes, that may contain the drive identifier, filename and extension. The filename is required. If you omit the drive identifier, BASIC assumes the current drive. If the filename is eight characters or fewer and you omit the extension, BASIC appends the extension **.BAS**.

The file must be in ASCII format, that is, it must have been SAVEd with the A option.

Program lines in the disk program are inserted into the resident program in sequential order. For example, suppose that three of the lines from the disk program are numbered 75, 85 and 90, and three of the lines from the current program are numbered 70, 80, and 90. When MERGE is used on the two programs, this portion of the new program is numbered 70, 75, 80, 85, 90.

If line numbers on the disk program coincide with line numbers in the resident program, the disk program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

## Example

Suppose you have a BASIC program on disk, PROG2.TXT (saved in ASCII), which you want to merge with the program you've been working on in memory. Then we use:

```
MERGE "PROG2.TXT"
```

merges the two programs.

## Sample Programs

MERGE provides a convenient means of putting program modules together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in memory:

```
80 REM                    MAIN PROGRAM
90 REM LINE NUMBER RESERVED FOR
   SUBROUTINE HOOK
100 REM                   PROGRAM LINE
110 REM                   PROGRAM LINE
120 REM                   PROGRAM LINE
130 END
```

And suppose the following subroutine, SUB.TXT, is stored on disk in ASCII format:

```
90 GOSUB 1000 SUBROUTINE HOOK
1000 REM                  BEGINNING OF
                          SUBROUTINE
1010 REM                  SUBROUTINE LINE
1020 REM                  SUBROUTINE LINE
1030 REM                  SUBROUTINE LINE
1040 RETURN
```

You can MERGE the subroutine with the main program with:

```
MERGE "SUB.TXT"
```

and the new program in memory is:

```
80   REM                  MAIN PROGRAM
90   GOSUB 1000 SUBROUTINE HOOK
100  REM                  PROGRAM LINE
110  REM                  PROGRAM LINE
120  REM                  PROGRAM LINE
130  END
1000 REM                  BEGINNING OF
                          SUBROUTINE
1010 REM                  SUBROUTINE LINE
1020 REM                  SUBROUTINE LINE
1030 REM                  SUBROUTINE LINE
1040 RETURN
```

# MID$ Statement

## MID$(*oldstring, position* [,*length*]) = *replacement string*

Replaces a portion of an *oldstring* with *replacement string*.

*Oldstring* is the variable name of the string you want to change.

*Position* is a number specifying the position of the first character to be changed.

*Length* is a number specifying the number of characters to be replaced.

*Replacement string* is the string to replace a portion of oldstring.

The length of the resultant string is always the same as the original string. If *replacement string* is shorter than *length*, the entire replacement string is used.

## Examples:

```
A$ = "LINCOLN"
MID$ (A$, 3, 4) = "12345": PRINT A$
```

returns LI1234N.

```
MID$ (A$, 5) = "01": PRINT A$
```

returns LINC01N.

```
MID$ (A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

# MID$ Function

## MID$(*string, integer* [*,number*])

Returns a substring of a string.

*Number* is the number of characters in the substring. It must be in the range 1 to 255.

*Integer* specifies the position in the string to begin returning characters from.

If you omit *number* or there are fewer than *number* characters to right of *integer* position, BASIC returns all right most characters, beginning with the character at position *integer*.

If *integer* is greater than the number of characters in *string*, MID$ returns a null string.

## Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

## Sample Program

```
200 INPUT "AREA CODE AND NUMBER
    (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$
    " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

# MKD$, MKI$, MKS$ Function

**MKI$(*integer expression*)**
**MKS$(*single-precision expression*)**
**MKD$(*double-precision expression*)**

Convert numeric values to string values.

Any numeric value placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These three functions are the inverse of CVD, CVI, and CVS. The byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

MKD$ returns an eight-byte string; MKI$ returns a two-byte string; and MKS$ returns a four-byte string.

## Example

```
LSET AVG$ = MKS$(0.123)
```

## Sample Program

```
1350 OPEN "R", 1, "TEST.DAT", 14
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKS$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1, 1
1410 CLOSE 1
```

For a program that retrieves the data from TEST.DAT, see CVD/CVI/CVS.

# NAME Statement

## NAME *old filespec* AS *new filespec*

Renames *old filespec* as *new filespec*.

With this statement, the data in the file is left unchanged. The *new filespec* may not contain a password or drive specification.

## Example

```
NAME "FILE.BAS" AS "FILE.OLD"
```

renames FILE.BAS as FILE.OLD.

# NEW

Deletes the program currently in memory and clears all variables.

NEW returns you to the command mode.

## Example

```
NEW
```

# OCT$ Function

## OCT$(*number*)

Computes the octal value of *number*.

OCT$ returns a string which represents the octal value of *number*. The value returned is like any other string — it cannot be used in a numeric expression.

## Examples

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the following strings:

```
36        62        132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

# ON COM(1) Communication Statement

## ON COM(1) GOSUB *line number*

Transfers program control to a subroutine beginning at *line number* when activity occurs on the communication channel.

*line number* is the first line of the subroutine to be executed when activity occurs on the communication channel. If you specify *line number* 0, you turn communication trapping off. It is the same as executing a COM(1) OFF statement.

The ON COM(1) statement is only executed if a COM(1) ON statement has been executed to enable communication trapping. If a COM(1) STOP statement has been executed to temporarily halt communication trapping, the subroutine is executed immediately after the next COM(1) ON statement is executed.

When the ON COM(1) statement is executed, BASIC immediately issues a COM(1) STOP statement to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another COM(1) ON statement to enable communication trapping again, unless the subroutine executes a COM(1) OFF statement.

## Example

```
10 COM(1) ON
.
.
.
.
200 ON COM(1) GOSUB 1000
```

Line 10 turns on communication trapping. After each program statement is executed, BASIC checks to see if the communication buffer contains characters. If it does, BASIC immediately executes the subroutine beginning at Line 1000.

If you execute a simple RETURN statement at the end of the subroutine, BASIC returns to the next statement after the statement that activated the trap. For example, if activity occurs while BASIC is executing Line 100, the RETURN returns to execute Line 110.

You may also use the RETURN *line number* option form of the RETURN statement. However, do so with care because any GOSUB, FOR, or WHILE statement remains active during trapping.

## Example

```
10 COM(1) ON
20 ON COM(1) GOSUB 1000
30 FOR I = 1 TO 10
40 PRINT I
50 NEXT I
.
.
1000 ' SUBROUTINE CODE
.
.
1050 RETURN 200
```

If activity occurs on the communication channel while the FOR/NEXT loop is executing, BASIC immediately executes the subroutine beginning at Line 1000. But the subroutine returns to Line 200 instead of completing the FOR/NEXT loop. This results in a "For without next" error because any GOSUB, FOR, or WHILE statement remains active during key trapping.

If the RETURN statement does not include a line number, program control returns to complete the FOR/NEXT loop, and no error occurs.

# ON ERROR GOTO **Statement**

## ON ERROR GOTO *line*

Transfers control to *line* if an error occurs.

This lets your program "recover" from an error and continue execution. (Normally, you have a particular type of error in mind when you use the ON ERROR GOTO statement).

ON ERROR GOTO has no effect unless it is executed before the error occurs. To disable it, execute an ON ERROR GOTO 0. If you use ON ERROR GOTO 0 inside an error-trapping routine, BASIC stops execution and prints an error message. If you have no recovery procedure for an error, ON ERROR GOTO 0 stops execution and prints an error message for the error that caused the trap.

**Note:** If an error occurs during execution of an error handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

The error-handling routine must be terminated by a RESUME statement. See RESUME.

## Example

```
10 ON ERROR GOTO 1500
```

branches program control to line 1500 if an error occurs anywhere after line 10.

For the use of ON ERROR GOTO in a program, see the sample program for ERROR.

# ON . . . GOSUB — Statement

## ON *number* GOSUB *line1, line2, . . .*

Branches to a subroutine at the *line* specified by the value of *number*.

*Number* must be between 0 and 255, inclusive. For example, if *number*'s value is three, the third line number in the list is the destination of the branch.

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

## Example

```
ON Y GOSUB 1000, 2000, 3000
```

if Y = 1, the subroutine beginning at 1000 is called. If Y = 2, the subroutine at 2000 is called. If Y = 3, the subroutine at 3000 is called.

## Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

# ON . . . GOTO Statement

## ON *number* GOTO *line1, line2, . . .*

Goes to the *line* specified by the value of *number*.

*Number* is a numeric expression between 0 and 255.

This statement is very similar to ON . . . GOSUB. However, instead of branching to a subroutine, it branches control to another program line.

The value of *number* determines to which line the program will branch. For example, if the value is four, the fourth line number in the list is the destination of the branch. If there is no fourth line number, control passes to the next statement in the program.

If the value of expression is negative or greater than 255, an "Illegal function call" error occurs. Any amount of line numbers may be included after GOTO.

## Example

```
ON MI GOTO 150, 160, 170, 150, 180
```

tells BASIC to "Evaluate MI,"
if the value of MI equals one then go to line 150;
if it equals two, then go to 160;
if it equals three, then go to 170;
if it equals four, then go to 150;
if it equals five, then go to 180;

if the value of MI doesn't equal any of the numbers one through five, advance to the next statement in the program".

# ON KEY GOSUB Statement

## ON KEY(*number*) GOSUB *line number*

Transfers program control to a subroutine when you press a function key or a cursor direction key.

*line number* is the first line number in the subroutine to execute when the specific key is pressed. If you specify a *line number* 0, you turn key trapping off for that key. It is the same as executing a KEY( ) OFF statement.

*number* may be a number in the range 1 to 16, indicating the number of the key to trap. Function keys use their corresponding function key number. The cursor direction keys are numbered:

| | |
|---|---|
| (↑) | 13 |
| (←) | 14 |
| (→) | 15 |
| (↓) | 16 |

The ON KEY( ) GOSUB statement is only executed if a KEY( ) ON statement has been executed to enable key trapping for that key. If a KEY( ) STOP statement has been executed to temporarily halt key trapping for that key, the subroutine is executed immediately after the next KEY( ) ON statement for that key is executed.

When the ON KEY( ) statement is executed, BASIC immediately issues a KEY( ) STOP statement for that key to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another KEY( ) ON statement for that key to enable key trapping again, unless the subroutine executes a KEY( ) OFF statement for that key.

If you execute a simple RETURN statement at the end of the subroutine, BASIC returns to the next statement after the statement that activated the trap. For example, if you press the specific key while BASIC is executing Line 100, the RETURN returns to execute Line 110.

You may also use the RETURN *line number* option form of the RETURN statement. Do so with care, however, because any GOSUB, FOR, or WHILE statement remains active during key trapping.

## Example

```
10 KEY(1) ON
20 ON KEY(1) GOSUB 1000
30 FOR I = 1 TO 10
40 PRINT I
50 NEXT I
.
.
1000 ' SUBROUTINE CODE
.
.
1050 RETURN 200
```

If you press Function Key 1 while the FOR/NEXT loop is executing, BASIC immediately executes the subroutine beginning at Line 1000. But the subroutine returns to Line 200 instead of completing the FOR/NEXT loop. This results in a "For without next" error because any GOSUB, FOR, or WHILE statement remains active during key trapping.

If the RETURN statement does not include a line number, program control returns to complete the FOR/NEXT loop and no error occurs.

# ON STRIG Statement

## ON STRIG (*integer*) GOSUB *line number*

Branches to a subroutine when you press the specified mouse button.

*integer* specifies the number of the button pressed. *integer* may be 0 for the left button and 1 for the right button.

*line number* is the first line number of the subroutine to be executed when you press the mouse button. Specifying a *line number* of 0 turns the trap off and is the same as executing a STRIG OFF statement.

The ON STRIG( ) GOSUB statement is only executed if a STRIG ON statement has been executed to enable mouse button trapping. If a STRIG STOP statement has been executed to temporarily halt mouse button trapping, the subroutine is executed immediately after the next STRIG ON statement is executed.

When the ON STRIG( ) GOSUB statement is executed, BASIC immediately issues a STRIG STOP statement to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another STRIG ON statement to enable mouse button trapping again, unless the subroutine executes a STRIG OFF statement.

If you execute a simple RETURN statement at the end of the subroutine, BASIC returns to the next statement after the statement that activated the trap. For example, if activity occurs while BASIC is executing Line 100, the RETURN returns to execute Line 110.

You may also use the RETURN *line number* option form of the RETURN statement. Do so with care because, however, any GOSUB, FOR, or WHILE statement remains active during trapping.

# Example

```
10 ON STRIG(0) GOSUB 1000
20 ON STRIG(1) GOSUB 2000
30 PRINT "Press one of the mouse buttons."
40 FOR I = 1 TO 3000:NEXT I
50 GOTO 30
1000 PRINT "You pressed the left button." :RETURN
2000 PRINT "You pressed the right button."
     :RETURN
```

Lines 10 and 20 turn on mouse button trapping. Line 30 prints a message for you to press one of the buttons. Line 40 waits for you to press a button. If you press the left button, BASIC transfers program control to the subroutine at Line 1000. If you press the right button, BASIC transfers program control to the subroutine at Line 2000. If you don't press a button, Line 50 returns to print the message again. This program is a continuous loop. To end the program, press **BREAK**.

# OPEN **Statement**

## OPEN *mode, buffer, filespec* [,*record length*]

## OPEN *filespec* [FOR *mode*] AS *buffer* [LEN = *record length*]

Establishes an input/output path for a file or device.

*buffer* is an integer in the range 1 to 15. It specifies the I/O buffer to use when accessing the file.

*filespec* specifies the device identifier, the filename, and the password. The password and device identifier are optional when you OPEN a disk file. If you omit the device identifier, BASIC assumes the current drive. Filename and password are optional with all other devices. You must enclose *filespec* in quotes.

*device identifier* indicates the physical device with which you want to communicate. Some devices restrict the direction of communication. These are the device identifiers and the *mode* with which they can be used:

| | |
|---|---|
| A: - D: | which disk drive to access. May be OPENed for all modes. |
| KYBD: | keyboard. INPUT only. |
| SCRN: | screen. OUTPUT only. |
| LPT1: | line printer 1. OUTPUT only. |
| LPT2: | line printer 2. OUTPUT only. |
| COM1: | RS232 communications 1. OUTPUT, INPUT, or RANDOM. |

*record length* is an integer in the range 2 to 32768 that sets the record length for random access files. It may not exceed the maximum set with /S: when you loaded BASIC. **Do not** use this option with sequential access files. If you omit *record length*, BASIC assumes a default record length of 128 byte.s

*mode* specifies any of the following:

| | |
|---|---|
| O or OUTPUT | sequential output mode |
| I or INPUT | sequential input mode |
| A or APPEND | sequential output and extend mode |
| R or RANDOM | direct input/output mode |

You must enclose *mode* in quotes in the first form of the syntax and you may only specify the abbreviated form of *mode*. If you omit *mode* in either form of the syntax, BASIC assumes random access.

In the second form of the syntax, you must specify the complete word for *mode*. You may not specify RANDOM. If you want to use random access in the second form of the syntax, omit *mode*.

If you OPEN a file for INPUT that does not exist, a "File Not Found" error occurs. If you OPEN a file for OUTPUT that does not exist, BASIC creates the file. If you OPEN a file for APPEND that does not exist, BASIC creates the file and sets the mode to RANDOM. If you OPEN a file for RANDOM access with a record length that does not match the record length assigned to the file when it was created, an error occurs.

You may OPEN a file for output in only one buffer at a time. Once you assign a buffer to a file with the OPEN statement, you cannot use that buffer in another OPEN statement until you close the first file. However, BASIC allows you to access the same file for input by opening it in different buffers. You may keep several records from the same file in memory for quick access.

## Examples

OPEN "R", 2, "TEST.DAT"
opens the file TEST.DAT in random access mode, using buffer 2. If TEST.DAT does not exist, BASIC creates it on the current drive. The record length is 128 bytes.

OPEN 1, "LIST.DAT", 80
opens the file LIST.DAT in random access mode, with a record length of 80.

OPEN "LPT1:" FOR OUTPUT AS #2
opens line printer 1 for sequential output using buffer 2.

OPEN "A:DATA.BAS" FOR INPUT AS #1
opens the file DATA.BAS on Drive A: for sequential input using buffer 1.

# OPEN "COM1: [*speed*] [,*parity*] [,*data*] [,*stop*] [,RS] [,CS[*seconds*] ] [,DS[*seconds*] ] [,CD[*seconds*] ] [,*mode*] [,LF]" AS [*buffer*] [LEN = *number*]

Opens a file and allocates a buffer for RS-232C (Asynchronous Communications Adapter) communication.

*speed* is an integer specifying the transmit and receive rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. If you omit *speed*, BASIC sets the *speed* at 300 bps.

*parity* is a constant specifying the parity to be used when the data is transmitted and received. The constant must be one of the following:

| | |
|---|---|
| E | indicates EVEN transmit parity, EVEN receive parity checking. |
| O | indicates ODD transmit parity, ODD receive parity checking. |
| M | indicates parity bit always transmitted and received as a mark (a 1 bit). |
| S | indicates parity bit always transmitted and received as a space (a 0 bit). |
| N | indicates no transmit parity, no receive parity checking. |

If you omit *parity*, BASIC assumes E (EVEN).

*data* is an integer specifying the number of transmit and receive bits. Valid values are 4, 5, 6, 7, and 8. If you do not specify *data*, BASIC assumes 7.

**Note:** Four *data* bits with **no** parity and eight *data* bits **with** *parity* are illegal.

*stop* must be either 1 or 2 to indicate the number of stop bits. If you omit *stop*, 75 and 100 bps transmit two *stop* bits, and all other speeds transmit one *stop* bit.

*buffer* is a number 1 through 15 indicating the buffer that accesses the file.

*number* specifies the maximum number of bytes that can be accessed in the communications buffer by GET and

PUT statements. If you omit the LEN option, BASIC assumes 128 bytes.

The parameters *speed, parity, data,* and *stop*, are all positional. That is, they must be in the order specified in the syntax. The remaining parameters are not positional. They may be in any order or you may omit them.

The remaining parameters control the software communication signal lines between two terminals. If you omit the CS, DS, or CD options, the signals are not checked at all. Only include these parameters if you are testing these software signals.

The RS option suppresses the Request To Send (RTS) signal. Request To Send is a signal that is sent from the sending terminal to the receiving terminal to ensure that the receiving terminal is ready to accept communication data. When you execute an OPEN "COM1: statement, the RTS line is turned on, unless you include the RS option.

The CS option controls the Clear To Send (CTS) signal which is sent from the receiving terminal to the sending terminal to let the sending terminal know that the receiving terminal is ready to receive.

You can think of RTS and CTS as a hand-shaking exercise, in which the two terminals let each other know that they are ready to send and/or receive data. RTS is an output signal from the sending terminal, and CS is an input signal to the sending terminal.

The DS option controls the Data Set Ready (DSR) signal. The DSR signal ensures that there is a data set, such as a modem, present to transmit the data.

The CS option controls the Carrier Detect (CD) signal. The CD signal is an input signal that ensures that the data set is ready to transmit the data.

The *seconds* argument in the CS, DS, and CD options specifies the number of milliseconds to wait for the signal before returning a "Device Timeout" error. *seconds* may

be in the range 0 to 65535. If you omit *seconds* or specify a zero, the signal is not checked at all.

If you specify RS, *seconds* default to zero for CS. If you omit RS, the default for CS is 1000. Either an RS or a CS is required. That is, if you omit RS, the Clear To Send signal is not checked. If you include RS, OPEN "COM1: waits 1 second for CS before issuing a "Device Timeout" error.

If you omit *seconds* after the DS option, the default value is 1000, and OPEN "COM1: waits 1 second before issuing a "Device Timeout" error. If you omit *seconds* after CD, the default is zero and the signal is not checked.

I/O statements to a communication file do not execute if these signals are off. The system waits one second before returning a "Device Timeout" error. Specifying these options allows you to ignore these signals or to specify the length of time to wait for the signal.

The LF option sends a line feed character after every carriage return. This is useful if you are printing the communication data to a serial line printer. A line feed is also sent after the carriage return that is the result of the width setting. Note that when you specify the LF option INPUT# and LINE INPUT# stop when they see a carriage return and ignore the line feed.

*mode* specifies the type of data that is transmitted. *mode* may be either BIN for binary mode or ASC for ASCII mode. If you omit *mode*, OPEN "COM1: opens the device in binary mode.

If you specify the BIN mode, OPEN "COM1: does not expand tabs to spaces, does not force a carriage return at the end of the line, does not recognize Control Z as an end-of-file, and ignores the LF option.

If you specify the ASC mode, OPEN "COM1: expands tabs to spaces, forces a carriage return at the end of the line, and recognizes Control Z as the end-of-file. When you close the channel, Control Z is sent over the RS-232C line.

# Examples

**OPEN "COM1:" AS 1**

opens File 1 for communication at a rate of 300 bps with even parity, seven data bits, and one stop bit. RTS signal is sent.

**OPEN "COM1: 9600, N,8,1,BIN" AS 2**

opens File 2 for communication at a rate of 9600 bps with no parity, 8 data bits, and 1 stop bit. The data is binary.

**OPEN "COM1: 4800,,,,CS3000,DS2000" AS 1**

opens File 1 for communication at a rate of 4800 bps with even parity, seven data bits, and one stop bit. RTS is sent. OPEN "COM1: issues "Device Timeout" error if there is no CS signal after 3 seconds and no DS signal after 2 seconds. Note that even though parity, data, and stop are not included, the commas are required.

# OPTION BASE **Statement**

## OPTION BASE *n*

Sets *n* as the minimum value for an array subscript.

*N* may be 1 or 0. The default is 0.

If you use this statement in a program, it must precede the DIM statement.

If the statement

```
OPTION BASE 1
```

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

# OUT **Statement**

## OUT *port, data byte*

Sends a *data byte* to a machine output *port*.

*Port* is an integer between 0 and 65535.

*Data byte* is an integer between 0 to 255.

A port is an input/output location in memory.

## Example

```
OUT 32,100
```

sends 100 to port 32.

# PAINT Statement

## PAINT [STEP] (*x-coordinate,y-coordinate*) [*color* [,*border*]]

Fills in an area on the display with a selected color.

*x-coordinate* indicates the x coordinate at which to begin. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4 *x-coordinate* may be in the range 0 to 640.

*y-coordinate* indicates the y coordinate at which to begin. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4 *y-coordinate* may be in the range 0 to 400.

*color* specifies a color number in the current palette. In Screen Mode 1, *color* may be in the range 0 to 3. In Screen Mode 3, *color* may be in the range 0 to 7. In Screen Modes 2 and 4, *color* may be either 0 or 1. If you omit *color* in Screen Modes 1 or 3, BASIC assumes color 3. If you omit *color* in Screen Modes 2 or 4, BASIC assumes white.

*border* specifies the color of the border of the object and must be a color number in the current palette. *border* may be specified in Screen Modes 1, 2, and 4 only. *border* is always black in Screen Mode 3. In Screen Mode 1, *border* may be in the range 0 to 3. In Screen Modes 2 and 4, *border* may be either 0 or 1. If you omit *border*, BASIC assumes the value of *color*, and the object has the same color border and center.

BASIC begins to change the color of points at the point you specify with x and y coordinates. BASIC continues to change the color of every point that is not the same color as *color*. When BASIC PAINTs one line of points without changing the color of any point in that line PAINT is complete.

PAINT must start on a non border point. If the point is already *border* or *color* color, BASIC does not execute the PAINT statement.

PAINT can fill any figure, but PAINTing "jagged" edges or very complex figures may result in an "Out of Memory" error. If this happens, you must use the CLEAR statement to increase the amount of stack space available.

## PALETTE [*position number, new color*]

Changes one of the colors in the current palette.

*position number* specifies which position in the current palette you want to change. *position number* may be a number in the range 0 to 3 in Screen Mode 1; and 0 to 7 in Screen Mode 3.

*new color* specifies the new color number you want in that position in the current palette. *new color* may be a number in the range −1 to 31. If you specify a value of −1 for any position number, that position number retains its default value and cannot be changed by subsequent PALETTE or PALETTE USING statements.

The PALETTE statement allows you to change the color in the current palette. The default values for Palettes 0 and 1 are

| Position Number | Palette 0 | Palette 1 |
|---|---|---|
| 1 | Green | Cyan |
| 2 | Red | Magenta |
| 3 | Yellow | White |
| 4 | White | Light Red |
| 5 | Light Cyan | Light Green |
| 6 | Light Blue | Light Blue |
| 7 | Light Yellow | Light Yellow |

When you select a palette, with a COLOR/Graphic statement, you tell BASIC to associate the position number with these colors when you use the number as the color parameter in graphics statements, such as LINE or PSET.

You can use the PALETTE statement to change the default values of the colors. You may change any of the position numbers to these color numbers.

| | |
|---|---|
| 0, 8, 16, or 24 | Black |
| 1 or 17 | Blue |
| 2 or 18 | Green |
| 3 or 19 | Cyan |
| 4 or 20 | Red |
| 5 or 21 | Magenta |
| 6 or 22 | Brown |
| 7 or 23 | Gray |
| 9 or 25 | Light Blue |
| 10 or 26 | Light Green |
| 11 or 27 | Light Cyan |
| 12 or 28 | Light Red |
| 13 or 29 | Light Magenta |
| 14 or 30 | Yellow |
| 15 or 31 | White |

For example, if you select Palette 0 with this statement

```
COLOR 0,0
```

number 1 is associated with green. You can use the PALETTE statement to change that value so that number 1 is associated with a different color from the list above. For example, we want to change number 1 to magenta. Use this statement

```
PALETTE 1,5
```

Number 1 in the current palette (Palette 0) changes from green to magenta.

When you execute a palette statement to change the default values, the new values remain in effect until you execute another COLOR/Graphic, PALETTE or PALETTE USING statement. A PALETTE statement without parameters forces the position numbers to return to their default values.

You can only change one position in the palette each time you execute a PALETTE statement. To change more than one position in a palette, see the PALETTE USING statement.

## Example

```
COLOR, 1:PALETTE 3,7
```

selects Palette 1 and changes the third position from yellow to gray.

```
PALETTE 1, -1
```

prevents position number 1 in the current palette from being changed by other PALETTE statements.

```
PALETTE
```

changes all positions in the current palette to their default values.

## Sample Program

```
10 COLOR 0,0
20 PALETTE 3,1
30 LINE (0,100) - (319,199),1
40 PAINT (1,100) 3,6
50 PALETTE
```

Line 10 selects Palette 0 as the current palette. Line 20 changes position number 3 from yellow to blue. Line 30 draws a vertical blue line across the center of the screen. Line 40 colors the bottom half of the screen light blue with a blue border. Line 50 causes the palette to return to its original value. Position 3 is now yellow again.

# PALETTE USING Graphics Statement

## PALETTE USING *array name* (*subscript*)

To change more than one of the color numbers in the current palette.

*array name* is the name of an integer array where you define the order of colors to be put in the current palette.

*subscript* is the position in the array that contains the value of the first position of the palette. BASIC assigns the remaining color numbers in the array to the palette in consecutive order. See the PALETTE statement for possible colors and their default values.

Load each element in the array with a color number. Group color numbers that you use together, consecutively in the array. For example, if you use both shades of blue with both shades of green, place their color numbers consecutively in the array, array A, like this

| Subscript | Color |
|---|---|
| 0 | 2 (green) |
| 1 | 10 (light green) |
| 2 | 1 (blue) |
| 3 | 9 (light blue) |

The statement

```
PALETTE USING A(0)
```

puts a 2 into position 0 in the current palette, 10 into position 1, 1 in position 2, and 9 into position 3.

The array may be larger than the palette. PALETTE USING stops filling the current palette when it reaches the last position in the palette. If you also use the two shades of blue with the two shades of red, and you also use the two shades of blue with the two shades of cyan, you can put the numbers for the shades of blue in your array as often as you need them. For example, you could expand the previous array to look like this

| Subscript | Color |
|---|---|
| 0 | 2 (green) |
| 1 | 10 (light green) |
| 2 | 1 (blue) |
| 3 | 9 (light blue) |
| 4 | 4 (red) |
| 5 | 12 (light red) |
| 6 | 9 (light blue) |
| 7 | 1 (blue) |
| 8 | 3 (cyan) |
| 9 | 11 (light cyan) |

To load the palette with the blues and reds use this statement:

```
PALETTE USING A(2)
```

Position 1 becomes number 1, blue. Position 2 becomes number 9, light blue. Position 3 becomes number 4, red. Position 4 becomes number 12, light red.

You could also load the palette with the blues and reds with this statement:

```
PALETTE USING A(4)
```

In this case, position 1 becomes number 4, red. Position 2 becomes number 12, light red. Position 3 becomes number 9, light blue. Position 4 becomes number 7, blue. The same colors are put into the palette, but in a different order.

To load the palette with the blues and cyans, use this statement:

```
PALETTE USING A(6)
```

BASIC starts loading the palette with the value of the sixth element in the array.

If you use the PALETTE statement to assign a value of −1 to a position in the palette, PALETTE USING does not change that position.

# PEEK **Function**

## PEEK (*memory location*)

Returns a byte from *memory location*.

The *memory location* must be in the range −32768 to 65535.

The value returned is an integer between 0 and 255. (For the interpretation of a negative value of *memory location*, see the statement VARPTR).

PEEK is the complementary function of the statement POKE.

## Example

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

# PLAY *string*

Plays musical notes specified by *string*.

*string* is a string expression consisting of one or more single character music commands.

The single character music commands are:

**A - G [#, +, −]**

The letters A through G play the notes of one musical scale. You may include an optional number sign (#) or plus (+) to indicate a sharp note or a minus (−) to indicate a flat note. You may only specify sharp or flat notes that correspond to the black keys on a piano. The letters A, C, D, F, and G may be followed by a plus because they are followed by black keys on a piano. The letters A, B, D, E, and G may be followed by minus because they are followed by black notes on a piano.

**L*integer***

Sets the length of the notes that follow. *integer* may be a value in the range 1 to 64. Here are a few of the more common lengths:

1 indicates a whole note.
2 indicates a half note.
4 indicates a quarter note.
8 indicates an eighth note.
16 indicates a sixteenth note.

If you only want to change the length for one note, *integer* may follow the note. For example, A16 is equivalent to L16A.

**O*integer***

Sets the current octave. There are 7 octaves, numbered 0 to 6. Each octave starts with C and ends with B. Octave 3 starts with middle C. If you omit *integer*, BASIC assumes Octave 4.

**N*integer***

Play a note. *integer* may be in the range 0 to 84. In the 7 possible octaves, there are 84 notes. Instead of specifying the letter and the octave of the note you may specify its number 1 to 84. Specifying an *integer* of zero means rest.

**P*integer***

Rest. *integer* may be in the range 1 to 64 and has the same meaning as *integer* with the L option.

**T*integer***

Sets the number of quarter notes in one minute. *integer* may be in the range of 32 to 255. If you omit *integer*, BASIC assumes 120 quarter notes in one minute. That is a moderato tempo. See the SOUND statement for information on beats per minute for common tempos.

With the O, N, P, and T commands, *integer* may also be a numeric variable in your BASIC program. Do not space between the command and the *integer* or between the command and the *variable*. You must include a semicolon after the variable name.

A dot after a note causes the note to play half again as long as the length specified by the integer with the L option. You may use more than one dot after each note. BASIC scales the length of time accordingly. Dots may also appear after the P option to scale the length of the rest.

**MF**

Sounds made by the PLAY and SOUND statements are to run in foreground. That is, each subsequent note or sound does not start until the previous note or sound is finished. If you omit MF or MB, BASIC assumes MF.

**MB**

Sounds made by the PLAY and SOUND statements are to run in background. That is, each note or sound is placed in a buffer allowing the BASIC program to continue execution while music plays in the "background." A maximum of 32 notes and/or rests can play in background at a time.

**MN**

Each note plays 7/8ths of the time specified by the L option. If you omit MN and MS, BASIC assumes MN.

**MS**

Each note plays 3/4ths of the time specified by the L option.

**X*variable*;**

Executes a substring. The X command lets you execute a second substring from a string, much like GOSUB. You can have one string execute another, which executes a third, and so on. *variable* is a string variable in your program that contains the substring you want to execute. *variable* may contain an X command to execute another substring. The semicolon after string is required.

## Example

```
10 PLAY "C4F4.C8F8.C16F8.G16A2F2"
20 INPUT "CAN YOU NAME THAT TUNE";A$
40 IF A$ = "THE EYES OF TEXAS" THEN GOTO
   50 ELSE PRINT "TRY AGAIN":GOTO 10
50 PRINT "THAT'S RIGHT!"
```

# POINT — Graphics Function

## POINT (*x-coordinate, y-coordinate*) = *variable*

Returns the color number of a point on the screen.

*x-coordinate* specifies the x coordinate of the point. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4, *x-coordinate* may be in the range 0 to 640.

*y-coordinate* specifies the y coordinate of the point. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4, *y-coordinate* may be in the range 0 to 400.

*variable* is numeric variable to hold the value returned by POINT.

The x and y coordinates must be absolute values. If you specify a point that is out of range, BASIC returns a −1.

If you are using either of the color graphics options, POINT returns the color number as it is defined in the current palette. In Screen Mode 1, POINT returns a value of 0 to 3. In Screen Mode 3, POINT returns a value 0 to 7. In Screen Modes 2 and 4, POINT returns a value of 0 or 1.

## Example

```
10 SCREEN 2
20 IF POINT(1,1)<>0 THEN PRESET (1,1) ELSE
   PSET (1,1)
```

If point 1,1 is any foreground color, PRESET changes it to the background color. If the point is the background color, PSET changes it to color number 3.

# POKE Statement

## POKE *memory location, data byte*

Writes *data byte* into *memory location*.

Both *memory location* and *data byte* must be integers. *Memory location* must be in the range −32768 to 65535.

POKE is the complementary statement of PEEK. The argument to PEEK is a memory location from which a byte is to be read.

PEEK and POKE are useful for storing data efficiently, loading assembly-language subroutines, and passing arguments (or results) to and from assembly-language subroutines.

## Example

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

# POS

**Function**

## POS(*number*)

Returns the position of the cursor.

*Number* is a dummy argument.

POS returns a number from 1 to 40 or 1 to 80, depending on the current width, indicating the current cursor-column position on the display.

## Example

```
PRINT TAB(40) POS(0)
```

prints 40. The PRINT TAB statement moves the cursor to position 40, therefore, POS(0) returns the value 40. (However, since a blank is inserted before the "4" to accommodate the sign, the "4" is actually at position 41).

## Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) > 70 THEN IF A$ = CHR$(32) THEN
    A$ = CHR$(13)
200 PRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that you cannot correct mistakes). Your computer keyboard is the typewriter keyboard. The program will keep watch at the end of a line so that no word is divided between two lines.

# PRINT **Statement**

## PRINT *data*, . . .

Prints numeric or string *data* on the display.

BASIC prints the values of the data items you list in this statement. If you omit *data*, BASIC prints a blank line.

You may separate the data items by commas, semicolons, or spaces. If you use commas, the cursor automatically advances to the next tab position before printing the next item. (BASIC divides each line into print zones containing 14 positions each, at columns 14, 28, 42, 56, and 70). If you use semicolons or spaces to separate the data items, PRINT prints the items without any spaces between them.

A semicolon or comma at the end of a line causes the next PRINT statement to begin printing where the last one left off. If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line. If the printed line is longer than 80 characters, BASIC continues printing on the next line.

Single-precision numbers with six or fewer digits that can be accurately represented in ordinary (rather than exponential) format, are printed in ordinary format. For example, 1E-7 is printed as .0000001; 1E-8 is printed as 1E-08.

Double-precision numbers with 16 or fewer digits that can be accurately represented in ordinary format, are printed using the ordinary format. For example, 1D-15 is printed as .000000000000001; 1D-16 is printed as 1D-16.

BASIC prints all numbers with a trailing blank, positive numbers with a leading blank, and precedes negative numbers with a minus sign.

To insert strings into this statement, surround them with quotation marks.

## Example

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES";
"BETWEEN"; "THESE"; "WORDS"
```

prints on the display: DONOTLEAVESPACESBETWEEN-THESEWORDS

## Sample Program

```
60 INPUT "ENTER THIS YEAR"; Y
70 INPUT "ENTER YOUR AGE";A
80 INPUT "ENTER A YEAR IN THE FUTURE";F
90 N = A + (F - Y)
100 PRINT "IN THE YEAR"F"YOU WILL
    BE"N"YEARS OLD"
RUN
```

Since F and N are positive numbers, PRINT inserts a space before and after them, therefore your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in line 100 by a comma,

```
100 PRINT "IN THE YEAR",F,"YOU WILL
    BE",N,"YEARS OLD"
```

BASIC would move to the next tab position after printing each data item.

# PRINT USING **Statement**

## PRINT USING *format*; *data item*, . . .

Prints *data items* using a *format* specified by you.

*Format* consists of one or more field specifier(s), or any alphanumeric character.

*Data item* may be string and/or numeric value(s).

This statement is especially useful for printing report headings, accounting reports, checks, or any other documents which require a specific format.

With PRINT USING, you may use certain characters, field specifiers, to format the field. These field specifiers are described below. They are followed by sample program lines and their output to the screen. You may use more than one field specifier, except as indicated.

# Specifiers for String Fields:

Print the first character in the string only.

```
PRINT USING "!"; "PERSONNEL"
P
```

\spaces\ — Print 2 + n characters from the string. If you type the backslashes without any spaces, BASIC prints two characters; with one space, BASIC prints three characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right.

```
PRINT USING "\ƀƀƀ\"; "PERSONNEL"
```

(three spaces between the backslashes)

```
PERSO
```

& — Print the string without modifications.

```
10 A$="TAKE":B$="RACE"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
TRACE
```

# Specifiers for Numeric Fields:

#  Print the same number of digit positions as number signs (#). If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). Numbers are rounded as necessary. You may insert a decimal point at any position. In that case, the digits preceding the decimal point are always printed (as zero, if necessary).

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding the number exceeds the field, a percent sign is also printed in front of the rounded number.

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

If the number of digits specified exceeds 24, an "Illegal function call" occurs.

```
PRINT USING "##.##";.75
0.75
```

```
PRINT USING "###.##";876.567
876.57
```

+  Print the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+##.##      ";
-98.45,3.50,22.22,-.9
-98.45     +3.50     +22.22     -0.90
```

```
PRINT USING "##.##+      ";
-98.45,3.50,22.22,-.9
98.54-     3.50+     22.22+     0.90-
```

(Note the use of spaces at the end of a format string to separate printed values).

**–** Print a negative sign after negative numbers (and a space after positive numbers). You may only use a negative sign to the right of a number.

```
PRINT USING "###.#-"; -768.660
768.7-
```

****** Fill leading spaces with asterisks. The two asterisks also establish two more positions in the field.

```
PRINT USING "**####"; 44.0
****44
```

**$$** Print a dollar sign immediately before the number. This specifies two more digit positions, one of which is the dollar sign. You may not use exponent format with $$.

```
PRINT USING "$$##.##"; 112.7890
$112.79
```

****$** Fill leading spaces with asterisks and print a dollar sign immediately before the number.

```
PRINT USING "**$##.##"; 8.333
***$8.33
```

Print a comma before every third digit to the left of the decimal point. The comma establishes another digit position.

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

^^^^ Print in exponential format. The four exponent signs are placed after the digit position characters. You may specify any decimal point position. You may not use $$ or **$ with exponent format.

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

_ Print next character as a literal character.

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

## Sample Program

```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P#
470 CLS : PRINT "PAY TO THE ORDER OF ";
480 PRINT USING "!! !! "; F$; "."; M$; ".";
490 PRINT L$
500 PRINT :PRINT USING A$; P#
```

In line 480, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the two spaces in "!!b̸!!b̸". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for format and P for item list in line 500. Any serious use of the PRINT USING statement would probably require the use of variables at least for item list rather than constants. (We've used constants in our examples for the sake of better illustration).

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,435.60 DOLLARS
```

# PRINT TAB                                                    Statement

## PRINT TAB(n)

Moves the cursor to the *n* position on the current line.

TAB may be used more than once in a print list.

Since numeric expressions may be used to specify a TAB position, TAB can be very useful in creating tables, graphs of mathematical functions, etc.

TAB can't be used to move the cursor to the left. If the cursor is to the right of the specified position, the TAB statement is simply ignored.

The first parenthesis must be typed immediately after the word TAB.

If *n* is greater than 80, BASIC divides *n* by 80 and uses the remainder of the division as the tab position. For example, if you enter the line:

```
PRINT "NAME"; TAB(84); "AMOUNT"
```

BASIC converts TAB(84) into TAB(4). Since the cursor is already at column five after printing NAME, BASIC moves the string AMOUNT to the next line. If, instead, you had typed TAB(85), BASIC would print AMOUNT on the same line.

If the string you are printing is too long to fit on the current line, BASIC moves the string to the next line.

## Example

```
PRINT TAB(5) "TABBED 5"; TAB(25) "TABBED 25"
```

Notice that no punctuation is needed after the TAB modifiers.

## Sample Program

```
220 CLS
230 PRINT TAB(2) "CATALOG NO."; TAB(16)
    "DESCRIPTION OF ITEM";
240 PRINT TAB(39) "QUANTITY"; TAB(51)
    "PRICE PER ITEM";
245 PRINT TAB(69) "TOTAL PRICE"
```

# PRINT# Statement

## PRINT# *buffer, item1, item2, . . .*

Prints data *items* in a sequential disk file.

*Buffer* is the buffer number used to OPEN the file for input.

When you first OPEN a file for sequential output, BASIC sets a pointer to the beginning of the file — that's where PRINT# starts printing the values of the *items*. At the end of each PRINT# operation, the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the display creates on the screen. For this reason, make sure to delimit the data so that it will be input correctly from the disk.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

When you include the USING option, data is written to the disk in the format you specify. See PRINT USING.

## Examples

```
If A = 123.45
PRINT# 1,A
```

writes this nine-byte character sequence onto disk:

b̸123.45b̸ carriage return

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example, if A = 2300 and B = 1.303, then

```
PRINT# 1, A,B
(ENTER)
```

writes the data on disk as

b̸ 2300 b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸ 1.303b̸ carriage return

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semicolons instead of commas.

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

writes the same data on disk as

123.45,1.303

An INPUT# statement reads this as two separate fields.

If string variables contain commas, semicolons, or leading blanks, write them to disk enclosed with quotation marks. For example, if A$ = CAMERA, AUTOMATIC and B$ = 102382, then

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

writes the data on disk as

CAMERAbbbbbbbbbbAUTOMATIC102382

An INPUT# statement reads this as two separate fields

```
A$ = CAMERA
B$ = AUTOMATIC102382
```

To separate these two strings properly on the disk, write double quotation marks to the disk using the hexadecimal character for quotation marks, CHR$(34).

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

writes the following image to disk

"CAMERA,AUTOMATIC""102382"

The statement

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

reads "CAMERA,AUTOMATIC" into A$ and "102382" into B$.

Files can be written in a carefully controlled format using PRINT# USING. You can also use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VAN", and C$ = "BEETHOVEN". Then the statement

```
PRINT# 1, USING"!.!.\bb\";A$;B$;C$
```

would write the data in nickname form:

L.V.BEET

(In this case, we didn't want to add any explicit delimiters.) See PRINT USING for more information on the USING option.

# PSET/PRESET Graphics Statement

## PSET [STEP] (*x-coordinate, y-coordinate*) [, *color*]
## PRESET [STEP] (*x-coordinate, y-coordinate*) [, *color*]

Draws a point on the display.

*x-coordinate* specifies the x coordinate of the point. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4, *x-coordinate* may be in the range 0 to 640.

*y-coordinate* specifies the y coordinate of the point. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4, *y-coordinate* may be in the range 0 to 400.

If you include the STEP option, the numbers you specify as coordinates are offsets from the most recent graphics point referenced. *x-coordinate* is the number of points in the horizontal direction, and *y-coordinate* is the number of points in the vertical direction. Precede the numbers with a plus ( + ) or minus ( – ) sign to indicate the direction (up, down, left, or right) from the most recent point referenced. The plus sign indicates to add the number to the most recent coordinante (right or up) and the minus indicates to subtract (left or down) the number from the most recent coordinate.

*color* specifies the color of the point and must be a color number in the current palette. In Screen Mode 1, *color* may be in the range 0 to 3. In Screen Mode 3, *color* may be in the range 0 to 7. In Screen Modes 2 and 4, *color* may be either 0 or 1.

The only difference between the PSET and PRESET statements is the default values for *color*. In Screen Modes 1 and 3, if you omit *color* with PSET, BASIC assumes a default value of 3. In Screen Modes 2 and 4, if you omit *color* with PSET, BASIC assumes white. If you omit *color* with PRESET, BASIC asssumes the background color for all Screen Modes and the point is invisible.

**Note:** BASIC does not print and does not issue an error message for points whose coordinate values are beyond the edge of the screen. However, values outside the integer range ( −32768 to 32767) cause an overflow error.

## Sample Program

```
10 FOR I=0 TO 100
20 PSET (I,I)
30 NEXT I        '(draw a diagonal line to (100,100))
40 FOR I=100 TO 0 STEP -1
50 PRESET (I,I),0
60 NEXT
70 '(clear out the line by setting each pixel to 0)
```

Lines 10 to 30 draw a diagonal line on the screen from the home position to position 100,100. Lines 40 to 60 erase the line by drawing another line at the same position in the background color.

# PUT — Statement

## PUT *buffer* [,*record*]

Puts a *record* in a direct-access disk file.

*Buffer* is the same buffer used to OPEN the file.

*Record* is the record number you want to PUT into the file. It is an integer between 1 and 65535. If you omit *record*, BASIC uses the current record.

This statement moves data from the buffer of a file into a specified place in the file.

If *record* is higher than the end-of-file record number, then *record* becomes the new end-of-file record number.

The first time you use PUT after OPENing a file, you must specify the *record*. The first time you access a file via a particular buffer, the next record is set equal to one. (The next record is the record whose number is one greater than the last record accessed).

See the chapter on "Disk Files" for programming information.

## Examples

```
PUT 1
```

writes the next record from buffer 1 to a direct-access file.

```
PUT 1, 25
```

writes record 25 from buffer 1 to a direct-access file.

# PUT — Communication Statement

## PUT *buffer, integer*

Transfers data from the file buffer to the communications buffer.

*buffer* must be the same *buffer* you assigned to the file in the OPEN "COM1: statement.

*integer* is the number of bytes to transfer from the file *buffer* into the communications buffer. *integer* cannot exceed the value you used in the LEN option in the OPEN "COM1: statement.

**Note:** Because of the low performance associated with telephone line communications, we recommend that you **not** use GET and PUT statements in such applications.

## Example

```
PUT 4,80
```

transfers 80 bytes from file buffer 4 to the communication buffer.

# PUT/Graphics — Graphics Statement

## PUT (*x-coordinate, y-coordinate*),*array* [,*action*]

Transfers an image stored in an array onto the screen.

*x-coordinate* indicates the x coordinate where the image begins. In Screen Mode 1, *x-coordinate* may be in the range 0 to 320. In Screen Modes 2, 3, and 4 *x-coordinate* may be in the range 0 to 640.

*y-coordinate* indicates the y coordinate where the image begins. In Screen Modes 1 and 2, *y-coordinate* may be in the range 0 to 200. In Screen Modes 3 and 4, *y-coordinate* may be in the range 0 to 400. If you omit *x-coordinate* and *y-coordinate*, BASIC begins the image at the last point referenced on the screen.

*array* is the array variable name that holds the image.

*action* may be PSET, PRESET, AND, OR, or XOR. *action* causes the transferred image to interact with the image already on the screen. If omit *action*, BASIC assumes XOR.

You use the GET/Graphics and PUT/Graphics statements together for animation and high-speed object motion in Screen Modes 1, 2, 3, or 4. The GET/Graphics statement transfers the screen image described by specified points of the rectangle into the array. The PUT/Graphics statement transfers the image from the array to the display.

The x and y coordinates specify the coordinate of the upper left corner of the image. An "Illegal Function call" error results if the image is too large to fit on the screen.

PSET transfers the data onto the screen exactly as it stored in the array.

PRESET produces an opposite image on the screen. In Screen Modes 2 and 4 an array value of 1 (white) becomes a 0 (black) on the screen. In Screen Modes 1 and 3, the color value in the array becomes the numeric opposite on the screen. For example, if the array contains a 0, that point becomes a 3 in Screen Mode 1 and a 7 in Screen Mode 3. These tables show the effects on color when you specify PRESET in Screen Modes 1 and 3.

| **Mode 1** | | **Mode 3** | |
|---|---|---|---|
| Array Color | Screen Color | Array Color | Screen Color |
| 0 | 3 | 0 | 7 |
| 1 | 2 | 1 | 6 |
| 2 | 1 | 2 | 5 |
| 3 | 0 | 3 | 4 |
| | | 4 | 3 |
| | | 5 | 2 |
| | | 6 | 1 |
| | | 7 | 0 |

AND transfers the image only if an image already exists at those points on the screen. If an image is on the screen, the new image is placed over the existing image. If an image is not on the screen and you specify AND as the *action*, BASIC does not execute the PUT/Graphics statement.

OR superimposes an image onto an existing image. OR transfers the image onto the screen whether or not an image already exists at that position.

XOR inverts the points on the screen where a point exists in the array image. When an image is PUT against a complex background twice, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

These tables show what effects AND, OR, and XOR have on color that is on the screen.

AND

ARRAY VALUE

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 2 | 2 | 0 | 0 | 2 | 2 |
| 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| 4 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 4 |
| 5 | 0 | 1 | 0 | 1 | 4 | 5 | 4 | 5 |
| 6 | 0 | 0 | 2 | 2 | 4 | 4 | 6 | 6 |
| 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

OR

ARRAY VALUE

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 1 | 3 | 3 | 5 | 5 | 7 | 7 |
| 2 | 2 | 3 | 2 | 3 | 6 | 7 | 6 | 7 |
| 3 | 3 | 3 | 3 | 3 | 7 | 7 | 7 | 7 |
| 4 | 4 | 5 | 6 | 7 | 4 | 5 | 6 | 7 |
| 5 | 5 | 5 | 7 | 7 | 5 | 5 | 7 | 7 |
| 6 | 6 | 7 | 6 | 7 | 6 | 7 | 6 | 7 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

XOR

ARRAY VALUE

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 |
| 2 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 |
| 3 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 |
| 4 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |
| 5 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 |
| 6 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 |
| 7 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

To perform object animation, follow these steps:

1. PUT the object on the screen using XOR.
2. Calculate the next position of the object.
3. PUT the object on the screen a second time at the previous location to remove the previous image.
4. Repeat step 1, PUTting the object at the next location.

If you do movement this way, the background is not changed. You can reduce flicker by minimizing the time between steps 4 and 1 and by ensuring enough time delay between 1 and 3. If you are animating more than one object, process every object at once, one step at a time.

If preserving the background is not important, you can perform animation using the PSET action verb. Leave a border around the image as large or larger than the maximum distance the object moves. When you move an object, this border effectively erases any points. This method may be faster than the method using XOR described above, since only one PUT is required to move an object.

# RANDOMIZE Function

## RANDOMIZE [*number*]

Reseeds the random number generator.

*number* is an integer in the range −32768 to 32767. If you omit *number*, BASIC suspends program execution and prompts you for a number before executing RANDOMIZE:

```
Random Number Seed (-32768 to 32767)?
```

If the random number generator is not reseeded, the RND function returns the same sequence of numbers each time it is executed. To change the sequence of random numbers every time the RND function is executed, place a RANDOMIZE statement before the RND function.

You can use the seconds digits of the TIME$ function to insure that the random number generator is reseeded with a different value each time BASIC executes the RANDOMIZE function. For example, the statement:

```
RANDOMIZE VAL(RIGHT$(TIME$,2) )
```

uses the seconds digits as the value of *number*. Because those digits are constantly changing, *number* has a different value each time BASIC executes this statement.

## Sample Program

```
10 CLS
20 RANDOMIZE VAL(RIGHT$(TIME$,2))
30 INPUT "PICK A NUMBER BETWEEN 1 AND
   100";A
40 B=INT(RND*100)
50 IF A = B THEN 80
60 PRINT "You lose, the answer is";B;"--try again."
70 GOTO 20
80 PRINT "You picked the right number -- you win."
```

## READ *variable*, . . . .

Reads values from a DATA statement and assigns them to *variables*.

BASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is used; the second time, the second value is used, and so on.

A single READ may access one or more DATA statements (each DATA statement is accessed in order), or several READs may access the same DATA statement.

The values read must agree with the variable types specified in list of variables, otherwise, a "Syntax error" occurs. If the number of variables in the READ statement exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed.

To reREAD DATA from the start, use the RESTORE statement. If the number of variables specified is lower than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element.

## Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable "T".

## Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40 PRINT "NAME", "AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$, AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERS, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, "END"
```

# REM

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you look at a listing of your program, or someone else does, it will be easier to figure it out.

If REM is used in a multi-statement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for REM.

## Sample Program

```
110 DIM V(20)
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
150 NEXT I
```

or

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

# RENUM Statement

## RENUM [*new line*] [*,line*] [*,increment*]

Renumbers a program, starting at *line*, using *new line* as the first new line and *increment* for the new sequence.

If you omit *new line*, BASIC starts numbering at line 10.
If you omit *line*, it renumbers the entire program.
If you omit *increment*, it increments each line by 10.

RENUM also changes all line number references appearing after ELSE, GOTO, GOSUB, THEN, ON ... GOTO, ON ... GOSUB, ON ERROR GOTO, RESUME, and ERL[relational operator].

## Examples

```
RENUM
```

renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

```
RENUM 600, 5000, 100
```

renumbers all lines 5000 to the end of the program. The first renumbered line becomes 600, and an increment of 100 is used between subsequent lines.

```
RENUM 10000, 1000
```

renumbers line 1000 and all higher-numbered lines. The first renumbered line becomes line 10000. An increment of 10 is used between subsequent line numbers.

```
RENUM 100, , 100
```

renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that you must include commas even though the middle argument is not included.

# Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10, 20 and 30, then the command:

   ```
   RENUM 15, 30
   ```

   is illegal, since the result would move thc third line of the program ahead of the second. In this case, an "Illegal Function Call" error occurs, and the original program is left unchanged.

2. RENUM will not create new line numbers greater than 65529. Instead, an "Illegal Function Call" error occurs, and the original program is left unchanged.

3. If an undefined line number is used inside your original program, RENUM prints a warning message, Undefined line XXXX in YYYY", where XXXX is the original line number reference and YYYY is the original number of the line containing XXXX. Note that RENUM renumbers the program in spite of this warning message. It does not change the incorrect line number reference, but it does renumber YYYY, according to the parameters in your RENUM command.

# RESET

Closes all open files on all drives.

If a diskette contains any open files, RESET rewrites the diskette's directory track.

RESET ensures that all files on all diskettes are closed before you remove them from the drives. RESET is the same as a CLOSE on each OPEN file.

# RESTORE Statement

## RESTORE [*line*]

Restores a program's access to previously-read DATA statements.

This lets your program re-use the same DATA lines.
If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

## Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA THIS IS THE FIRST ITEM, AND THIS IS
    THE SECOND
```

When this program is run,

```
THIS IS THE FIRST ITEM      THIS IS THE
FIRST ITEM
```

is printed on the display. Because of the RESTORE statement in line 170, the second READ statement starts over with the first DATA item.

# RESUME — Statement

## RESUME [*line*]
## RESUME NEXT

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME 0 both cause the computer to return to the statement in which the error occurred.

RESUME *line* causes the computer to branch to the specified line number.

RESUME NEXT causes the computer to branch to the statement following the point at which the error occurred.

A RESUME that is not in an error-handling routine causes a "RESUME without error" message.

## Examples

```
RESUME
```

if an error has occurred, this line transfers program control to the statement in which it occurred.

```
RESUME 10
```

if an error has occurred, transfers control to line 10.

## Sample Program

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

# RETURN **Statement**

## RETURN [*line number*]

Returns control to the line immediately following the most recently executed GOSUB.

*line number* is an optional parameter that you may include to return program control to a specific *line number* instead of the *line number* immediately following the GOSUB.

Use the *line number* parameter with caution. Any other GOSUB, WHILE, or FOR statement remains active while a GOSUB subroutine is executing. If you RETURN to a *line number* that does not complete these loops you get an error.

## Example

```
10 FOR I = 1 TO 3
20 PRINT I: GOSUB 100
30 NEXT I
40 PRINT J
100 'SUBROUTINE BEGINS HERE
110 RETURN 40
```

When I is equal to 1, 2, or 3 Line 110 causes a "FOR without NEXT" error because the FOR/NEXT loop has not been completed.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

## Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF
    A CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS";
    R
350 GOSUB 370
360 PRINT "AREA IS" ; A: END
370 A = 3.14 * R * R
380 RETURN
```

# RIGHT$ Function

## RIGHT$(*string, number*)

Returns the rightmost *number* characters of *string*.

RIGHT$ returns the last *number* characters of *string*. If LEN (string) is less than or equal to *number*, the entire string is returned.

## Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

## Sample Program

```
850 RESTORE : ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2), : GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY,
    SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE,
    BROOKLYN, NY"
910 DATA "HAMMON MANUFACTURING
    COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

# RND Function

## RND (*number*)

Generates a pseudorandom number between 0 and 1.

*Number* is an integer in the range −32767 to 32768.

Only zero has any effect on the random number that BASIC generates.

RND produces a pseudorandom number using the current "seed" number. BASIC generates the seed internally, therefore, it is not accessible to the user. RND produces the same sequence of random numbers each time the program is run unless you execute a RANDOMIZE statement to reseed the random number generator.

If you specify negative values for *number*, RND starts the sequence of random numbers at the beginning. RND(0) repeats the *last* number generated. If you omit *number*, or you specify a positive value, RND returns the next number in the sequence.

## Examples

```
PRINT RND(0)
```

prints a decimal fraction between 0 and 1

```
PRINT RND(1)
```

prints the next decimal fraction in the sequence.

### Sample Program

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
```

This program produces 5 random integers. Line 20 converts the decimal fraction returned by RND to a real number and truncates the real number to an integer.

# RSET **Statement**

## RSET *field name = data*

Sets *data* in a direct-access buffer *field name* in preparation for a PUT statement.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

# RUN    Statement

## RUN [*line*]
## RUN *filespec* [,R]

Executes a program.

RUN followed by a *line* or nothing at all simply executes the program in memory, starting at *line* or at the beginning of the program.

RUN followed by a *filespec* deletes the current contents of memory, loads a program from disk and then executes it. If *filespec* contains fewer than eight characters and you do not include an extension, BASIC appends the extension **.BAS**. Any resident BASIC program is replaced by the new program.

Option R leaves all previously OPEN files open. If omitted, BASIC closes all open files.

RUN automatically CLEARS all variables. However, it does not re-set the value of an ERL variable.

## Examples

```
RUN
```

starts execution at lowest line number.

```
RUN 100
```

starts execution at line 100.

```
RUN "PROGRAM.A"
```

loads and executes PROGRAM.A.

```
RUN "EDITDATA", R
```

loads and executes EDITDATA, leaving OPEN files open.

# SAVE  Statement

## SAVE *filespec* [,A] [,P]

Saves a program in a disk file under *filespec*.

*filespec* is a string expression that may contain the drive identifier and filename. If you omit the drive identifier, BASIC assumes the current drive. The filename is required. If the filename is eight characters or fewer and you do not include an extension, BASIC appends the extension **.BAS**. If *filespec* already exists, its contents are lost as the file is re-created.

SAVE without the A option saves the program in a compressed format. This takes up less disk space. It also helps in performing SAVEs and LOADs faster. BASIC programs are stored in RAM using compressed format.

Using the A option causes the program to be saved in ASCII format. This takes up more disk space. However, the ASCII format allows you to MERGE this program later on. Also, data programs which are read by other programs must usually be in ASCII.

If you use the A option, make sure your program doesn't have any embedded line feeds; otherwise, the computer will not be able to read it properly. Embedded line feeds are produced by pressing CTRL J simultaneously when typing a program line.

For compressed-format programs, a useful convention is to use the extension .BAS. For ASCII-format programs, use .TXT.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it fails. The only operations that can be performed on a protected file are: RUN, LOAD, MERGE, and CHAIN.

## Examples

SAVE "A:FILE1.BAS"

saves the resident BASIC program in compressed format. The file name is FILE1; the extension is .BAS. The file is placed on Drive A:.

SAVE "MATHPAK.TXT", A

saves the resident program in ASCII form, using the name MATHPAK.TXT, on the current drive.

# SCREEN Function

## *variable* = SCREEN (*row, column*),[1]

Returns the ASCII code or the color attribute for the character at the specified row and column.

*row* is an integer in the range 1 to 25.

*column* is an integer in the range 1 to 40 or 1 to 80, depending on the screen width.

The *1* indicates return the color attribute rather than the ASCII code.

SCREEN stores the ASCII character or the color attribute of the character at the specified *row* and *column* position in *variable*.

## Sample Program

```
10 LOCATE 20,20
20 PRINT "Robbie"
30 A = SCREEN(20,20):B = SCREEN(20,21)
40 PRINT A,B
```

Line 10 positions the cursor to row 20, column 20. Line 20 prints the message at the current cursor position. Line 30 stores the ASCII code for "R" in the variable A and the ASCII code for "o" in variable B. Line 40 prints

```
82        79
```

# SCREEN — Statement

## SCREEN *mode* [, *burst*]

Sets the screen attributes to be used by all other graphics statements.

*mode* is an integer in the range 0 to 4. Valid modes are:

0 — text mode at the current width (40 or 80).
1 — 320 x 200 medium resolution color graphics mode. You can only use WIDTH 40.
2 — 640 x 200 high resolution monochrome mode at the current width (40 or 80).
3 — 640 x 400 highest resolution color graphics mode at the current width (40 or 80).
4 — 640 x 400 highest resolution monochrome mode at the current width (40 or 80).

If you have the TV/Joystick option you may use Mode 0 in width 40 and Mode 1. If you have the High Resolution Monochrome Graphics Option, you may use modes 0, 1, 2 and 4. If you have the High Resolution Color Graphics Option, you may use all of the modes in all widths.

*burst* may be zero to disable color burst or any other number to enable color burst. If you specify zero, you can only display black and white images with subsequent graphics commands. *burst* is only valid with the color graphics options.

The SCREEN statement controls all graphics statements: CIRCLE, LINE, DRAW, POINT, PSET, PRESET, PALETTE, and PALETTE USING. When you select *mode* with the SCREEN statement, you set the valid coordinates and the number of colors that these statements may use.

If the SCREEN statement changes the *mode*, BASIC stores the new screen mode, erases the video display, sets the foreground color to white, and the background and border colors to black.

We recommend that you use these statements at the beginning of programs that you intend to run on machines that could have either graphics board:

```
SCREEN 0,0
WIDTH 40
```

For more information on the graphics statements refer to Chapter 6, "Introduction to Graphics."

## Examples

```
10 SCREEN 0,1
```

Selects text mode with color.

```
60 SCREEN 2
```

Changes to high resolution monochrome graphics mode.

## Sample Program

```
10 BLANK$=CHR$(32)
                    'Define blank character
20 CLS:WIDTH 40:SCREEN 1
                    'Double size characters
30 FORE=16:BACK=7:GOSUB 1000
                    'Blinking reverse video
40 FORE=0:GOSUB 1000
                    'Reverse video
50 BACK=0:GOSUB 200
                    'Background color black
60 WIDTH 80:SCREEN 0
                    'Normal size characters
```

```
70 BACK=7:FORE=16:GOSUB 1000
              'Blinking reverse video
80 FORE=0:GOSUB 1000
              'Reverse video
90 BACK=0:GOSUB 200
              'Background black
100 END
200 FORE=31:GOSUB 1000
              'Bright white blinking
210 FORE=18:GOSUB 1000
              'White blinking
220 FORE=17:GOSUB 1000
              'Underlined bright white blinking
230 FORE=10:GOSUB 1000
              'Bright white
240 FORE=9:GOSUB 1000
              'Underlined white
250 FORE=2:GOSUB 1000
              'White
260 RETURN
1000 PRINT    'Carriage return
1010 FOR I=1 TO 1000:NEXT I
              'Wait loop
1020 FOR CHAR=1 TO 6
              'Char = Graphics character to print
1030 COLOR FORE,BACK
              'Print in specified color
1040 PRINT CHR$(CHAR);BLANK$;
              'Print the character and a blank
1050 PRINT CHR$(CHAR+14);BLANK$;
1060 PRINT CHR$(CHAR+21);BLANK$;
1070 NEXT CHAR
1080 RETURN
```

This program prints graphics characters in all possible color combinations in normal mode and double size mode.

# SGN — Function

## SGN(*number*)

Determines *number's* sign.

If *number* is a negative number, SGN returns −1.
If *number* is a positive number, SGN returns 1.
If *number* is zero, SGN returns 0.

## Example

```
Y = SGN(A * B)
```

determines what the sign of the expression A * B is, and passes the appropriate number (−1,0,1) to Y.

## Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

# SIN **Function**

## SIN(*number*)

Computes the sine of *number*.

*Number* must be in radians. To obtain the sine of *number* when *number* is in degrees, use SIN(number * .01745329). The result is always single precision.

## Example

```
PRINT SIN(7.96)
```

prints .994385.

## Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN(A * .01745329)
```

# SOUND **Statement**

## SOUND *tone, duration*

Generates a sound with the *tone* and *duration* specified.

*Tone* is an integer in the range 37 and 32767 indicating the frequency in Hertz. Thirty-seven produces the lowest tone and 32767 produces the highest tone.

*Duration* is an integer in the range 0 to 65535 specifying the duration in clock ticks. Clock ticks occur 18.2 times per second. One produces the shortest sound and 65535 produces the longest sound.

While a SOUND statement produces noise, the program continues to execute. If another SOUND statement is encountered while the previous SOUND statement is still making noise, the program waits until the first sound ends before executing the next SOUND statement. However, if the *duration* of the new SOUND statement is zero, the previous SOUND statement is turned off. See the PLAY statement for more information about executing program lines during SOUND.

This statement can be especially useful in educational applications. For example, you can have the computer respond with a sound if a user has answered a program's prompt incorrectly (or vice versa).

You can use the SOUND or PLAY statements to generate musical notes from your computer. This chart shows the frequency you should specify to generate the notes in the octave above middle C. Middle C is the first note in the chart.

| Note | Frequency |
|---|---|
| C | 523.25 |
| D | 587.33 |
| E | 659.26 |
| F | 698.46 |
| G | 783.99 |
| A | 880.00 |
| B | 987.77 |
| C | 1046.50 |

To generate notes that are in the octave below middle C, find the frequency of the notes letter in the chart and divide that number by 2. For example, the note A in the octave below middle C has a frequency of 440.00

To generate notes that are in the octave above middle C, find the frequency of the notes letter in the chart and multiply that number by 2. For example, the note A in the octave above middle C has a frequency of 1760.00.

There are 1092 clock ticks per minute. To determine the number of clock ticks for one beat, divide the beats per minute into 1092. The chart below shows the number of clock ticks for some typical tempos.

| Tempo | Beats per Minute | Ticks per Minute |
|---|---|---|
| Largo | 40-60 | 27.3 -18.2 |
| Largehetto | 60-66 | 18.2 -16.55 |
| Adagio | 66-76 | 16.55-14.37 |
| Andante | 76-108 | 14.37-10.11 |
| Moderato | 108-120 | 10.11- 9.1 |
| Allegro | 120-168 | 9.1 - 6.5 |
| Presto | 168-208 | 6.5 - 5.25 |

## Sample Program

```
10 INPUT "IN HONOR OF WHOM WAS THE
   CONTINENT OF AMERICA NAMED"; A$
20 IF A$="AMERIGO VESPUCCI" THEN SOUND
   32000,200 ELSE GOTO 40
30 PRINT "THAT'S RIGHT!": END
40 SOUND 37,2 : PRINT "THE CORRECT
   ANSWER IS AMERIGO VESPUCCI"
```

# SPACE$ Function

## SPACE$(*number*)

Returns a string of *number* spaces.

*Number* must be in the range 0 to 255.

## Example

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE"
SPACE$(9) "QUANTITY"
```

prints DESCRIPTION, four spaces, TYPE, nine spaces, QUANTITY.

## Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```

# SPC **Function**

## SPC(*number*)

Prints *number* blanks.

*Number* is in the range 0 to 255. SPC does not use string space. The left parenthesis must immediately follow SPC.

SPC may only be used with PRINT, LPRINT, or PRINT# .

## Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints

```
HELLO               THERE
```

# SQR **Function**

## SQR(*number*)

Calculates the square root of *number*.

The *number* must be greater than zero.

The result is always single precision.

## Example

```
PRINT SQR(155.7)
```

prints 12.47798

## Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

# STICK Statement

## STICK (*integer*) = *variable*

Returns the number of points moved along the x and y axes since the last STICK statement.

*integer* may be zero to return the number of points moved on the x axis or one to return the number of points moved on the y axis.

*variable* is a numeric variable to hold the value returned by *integer*.

If you specify zero for integer, the following values are returned:

positive values, which indicate the number of points moved right on the x axis since the last STICK statement.

negative values, which indicate the number of points moved left on the x axis since the last STICK statement.

a zero, which indicates that no movement occurred on the x axis since the last STICK statement.

If you specify one for integer, the following values are returned:

positive values, which indicate the number of points moved down on the y axis since the last STICK statement.

negative values, which indicate the number of points moved up on the y axis since the last STICK statement.

a zero, which indicates that no movement occurred on the y axis since the last STICK statement.

# Example

```
10 STICK(0)= XMOVE:STICK(1)= YMOVE
20 IF STICK(0) < 0 THEN DIR$ = "left" ELSE DIR$
   = "right"
30 IF STICK(0) = 0 GOTO 40 ELSE PRINT "You
   moved" ABS(XMOVE) "pixels" DIR$ "and ";
40 PRINT "You didn't move any pixels on the x axis
   and ";
50 IF STICK(1) < 0 THEN DIR$ = "up" ELSE DIR$
   = "down"
60 IF STICK(1) = 0 GOTO 70 ELSE PRINT "you
   moved" ABS(YMOVE) "pixels" DIR$ "."
70 PRINT "you didn't move any pixels on the y axis."
```

Line 10 stores the number of points moved on the x axis in XMOVE and stores the number of points moved on the y axis in YMOVE. DIR$ is a string variable to store the direction that was moved. If the number returned by the STICK function is negative, the direction is left or up. If the number returned is positive, the direction is right or down. Lines 20-70 test the values returned by the two STICK functions and print a message to tell how many points you moved on each axis or that you didn't move any points.

# STOP

Stops program execution.

When a program encounters a STOP statement, it prints the message BREAK IN, followed by the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

The CONT command resumes execution at the point it was halted. But if the program itself is altered during the break, CONT cannot be used.

Unlike the END statement, STOP does not close files.

## Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number between 1 and 10 is assigned to X, then program execution halts at line 2270. You can now examine the value X with PRINT X. Type CONT to start the cycle again.

# STR$ Function

## STR$(*number*)

Converts *number* into a string.

If *number* is positive, STR$ places a blank before the string.

While arithmetic operations may be performed on *number*, only string functions and operations may be performed on the string.

## Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S$.

## Sample Program

```
10 A = 1.6 : B# = A : C# = VAL(STR$(A))
20 PRINT "REGULAR CONVERSION" TAB(40)
   "SPECIAL CONVERSION"
30 PRINT B# TAB(40) C#
```

# STRIG/Function Enable **Statement**

**STRIG ON**
**STRIG OFF**

Enables the STRIG Function command.

## STRIG ON

When you load BASIC, the default is STRIG OFF and you cannot execute STRIG/Function statements. STRIG ON lets you execute STRIG/Function statements to return the status of the mouse buttons. If you attempt to execute a STRIG/Function statement before you execute a STRIG ON statement, BASIC issues an "Illegal function call" error.

## STRIG OFF

If you execute a STRIG OFF statement you may not execute a STRIG/Function statement. Executing a STRIG/ Function statement after a STRIG OFF statement results in an "Illegal function call" error.

You cannot place a STRIG/Function statement in a subroutine that you branch to as a result of an ON STRIG( ) GOSUB statement. BASIC does not keep track of which button was pressed after the ON STRIG( ) GOSUB statement is executed. If you wish to trap both buttons and perform a different procedure for each button, you must execute a STRIG/Trap Enable for each button and you must branch to different subroutines with different ON STRIG( ) GOSUB statements.

See STRIG Function, STRIG/Trap Enable and ON STRIG( ) GOSUB for additional information on the mouse button trapping.

# STRIG/Trap Enable Statement

**STRIG(*integer*) ON**
**STRIG(*integer*) OFF**
**STRIG(*integer*) STOP**

Turns on, turns off, or temporarily halts mouse trapping.

*integer* is a value of 0 or 2 to indicate the mouse button you are trapping. 0 indicates the left button and 2 indicates the right button.

# STRIG( ) ON

STRIG( ) ON enables mouse trapping with the ON STRIG( ) GOSUB statement. If you execute a STRIG( ) ON statement, BASIC checks after every program statement to see if you pressed the mouse buttons. If you press the mouse buttons, BASIC transfers program control to the line number specified in the ON STRIG( ) GOSUB statement. See ON STRIG( ) GOSUB.

**Note:** Do not confuse the STRIG/Trap Enable statement with the STRIG/Function Enable statement. These are two separate statements that perform two distinct functions in BASIC.

# STRIG( )STOP

STRIG STOP temporarily halts mouse trapping. If you press the mouse buttons after a STRIG STOP statement is executed, BASIC does not transfer program control to the subroutine until mouse trapping is turned on again with a STRIG ON statement. BASIC remembers that the mouse buttons were pressed and transfers program control to the subroutine immediately after mouse trapping is turned on again.

# STRIG OFF

STRIG( ) OFF turns off mouse button trapping with the ON STRIG( ) GOSUB statement.

When you load BASIC, STRIG( ) OFF is the default because STRIG trapping slows program execution. Therefore, if you execute a STRIG( ) ON statement to enable mouse button trapping, we recommend that you also execute a STRIG( ) OFF statement when you no longer need to check for mouse button activity.

If you press the mouse buttons after a STRIG OFF statement is executed, BASIC does not remember that the mouse buttons were pressed when mouse trapping is turned on again.

## Example

```
10 STRIG(0) ON:STRIG(2) ON
20 ON STRIG(0) GOSUB 1000
30 ON STRIG(2) GOSUB 2000
```

Line 10 turns on mouse button trapping. When you press the left mouse button, BASIC transfers program control to the subroutine beginning at Line 1000. If you press the right mouse buttons, BASIC transfers program control to the subroutine beginning at Line 2000.

# STRIG **Function**

***variable* = STRIG *integer***

Returns the status of mouse buttons.

*integer* is a number in the range 0 to 3 to test the status of the mouse buttons.

*variable* is a numeric variable to recieve the value returned by *integer*.

Each *integer* tests for a different status of the two buttons and returns a numeric value in *variable* regarding the results of the test. The *integers* and their functions are

0 Tests to see if the left button has been pressed and released since the last STRIG/Function statement was executed. If the left button has been pressed since the last test, BASIC returns a −1 in *variable*. If the button has not been pressed, BASIC returns a 0.

1 Tests to see if you are currently pressing the left button. If you are pressing the left button, BASIC returns a −1 in *variable*. If you are not pressing the button, BASIC returns a 0.

2 Tests to see if the right button has been pressed and released since the last STRIG/Function statement was executed. If the right button has been pressed and released since the last test, BASIC returns a −1 in *variable*. If the button has not been pressed and released, BASIC returns a 0.

3 Tests to see if you are currently pressing the right button. If you are currently pressing the right button, BASIC returns a −1 in *variable*. If you are not pressing the button, BASIC returns a 0.

You must execute a STRIG/Function Enable statement before you can execute a STRIG/Function statement. If you attempt to execute a STRIG/Function statement before you execute a STRIG/Function Enable statement, BASIC issues an "Illegal function call" error. See STRIG/Function Enable Trap.

You cannot place a STRIG/Function statement in a subroutine that you branch to as a result of an ON STRIG( ) GOSUB statement. BASIC does not keep track of which button was pressed after the ON STRIG( ) GOSUB statement is executed. If you wish to trap both buttons and perform a different procedure for each button, you must execute a STRIG/Trap Enable for each button and you must branch to different subroutines with different ON STRIG( ) GOSUB statements.

## Example

```
10 STRIG ON:PRINT "Press one of the mouse
   buttons."
20 FOR I = 1 TO 1000:NEXT I
30 STAT0 = STRIG(0):STAT1 = STRIG(1)
40 STAT2 = STRIG(2):STAT3 = STRIG(3)
50 IF STAT0 = -1 THEN PRINT "You pressed the
   left button."
60 IF STAT1 = -1 THEN PRINT "You are still
   pressing the left button."
70 IF STAT2 = -1 THEN PRINT "You pressed the
   right button."
80 IF STAT3 = -1 THEN PRINT "You are still
   pressing the right button."
90 IF STAT0 = 0 AND STAT1 = 0 AND STAT2 =
   0 AND STAT3 = 0 THEN PRINT "Aren't you
   going to press a button?":GOTO 20
```

Line 10 enables the mouse function and prints a message telling you to press one of the mouse buttons. Line 20 gives you time to press one of the mouse buttons. Lines 30 and 40 check to see if either button has been pressed or is currently being pressed. Lines 50-90 print a message reporting the status of the buttons. If the buttons weren't pressed, Line 80 prints a message, and the program loops to Line 10 to start again. To end this program, press (BREAK).

# STRING$ Function

## STRING$(*number,character*)

Returns a string of *number* characters.

*Number* must be in the range 0 to 255.

*Character* is a string or an ASCII code. If you use a string constant, it must be enclosed in quotes. All the characters in the string have either the ASCII code specified, or the first letter of the string specified.

STRING$ is useful for creating graphs or tables.

## Examples:

```
B$ = STRING$(25, "X")
```

puts a string of 25 "X"s into B$.

```
PRINT STRING$(50, 10)
```

prints 50 blank lines on the display, since 10 is the ASCII code for a line feed.

## Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN THREE NUMBERS
     BETWEEN 33 AND 159"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20,
     N1): NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2):
     NEXT J
1080 PRINT STRING$(80, N3)
```

This program prints three strings. Each string has the character corresponding to one of the ASCII codes provided.

# SWAP **Statement**

## SWAP *variable1, variable2*

Exchanges the values of two variables.

Variables of any type may be SWAPed (integer, single precision, double precision, string). However, both must be of the same type, otherwise, a "Type mismatch" error results.

Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not been assigned values, an "Illegal Function Call" error results.

## Example

```
SWAP F1#, F2#
```

swaps the contents of F1# and F2#. The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

## Sample Program

```
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
```

## SYSTEM

Returns you to MS-DOS level.

Your resident BASIC program is not retained in memory.

**Note:** You cannot call DEBUG from BASIC.

## Examples

```
SYSTEM
```

returns you to MS-DOS. Your resident BASIC program is lost.

# TAB **Function**

## TAB(*number*)

Spaces to position *number* on the display.

*Number* must be in the range 1 to 255.

If the current print position is already beyond space *number*, TAB goes to that position on the next line. Space one is the leftmost position; the width minus one is the rightmost position.

TAB may only be used with the PRINT and LPRINT statements.

## Sample Program

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

The display shows:

```
NAME                     AMOUNT

G.T.JONES                $25.00
```

# TAN **Function**

## TAN(*number*)

Computes the tangent of *number*.

*Number* must be in radians. To obtain the tangent of *number* when it is in degrees, use TAN (number * .01745329). The result is always single precision.

## Example

```
PRINT TAN(7.96)
```

prints −9.396959

## Sample Program

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

# TIME$ Statement

***variable*= TIME$**

**TIME$ = "*string*"**

Sets or retrieves the current time.

*variable* is a variable in your BASIC program that receives the current time.

*string* is a literal, enclosed with quotes, that sets the time by assigning its value to TIME$.

You set the time in the format hh:mm:ss, where hh is the hours, mm is the minutes, and ss is the seconds. BASIC uses a 24 hour clock. For example, it sets 8:15 P.M. as 20:15:00.

## Setting the Time

*hh* may be any number 0 through 23.

*mm* and *ss* may be any number 0 through 59. If you omit the minutes, minutes **and** seconds default to zero. If you omit the seconds, seconds default to zero.

Although you may omit leading zeros in each of the values, you must include at least one digit of the previous value. For example, you may type 0:5 to set the time to 12:05 a.m. However, :5 is invalid.

## Retrieving the Time

BASIC always returns the time in the eight character (hh:mm:ss) format, with leading zeros. The time may be set by the operator prior to entering BASIC. If the operator did not set the time at the MS-DOS time prompt and the time was not set with the TIME$ statement, BASIC returns the length of the time that has elapsed since the terminal was powered on.

# Examples

TIME$ = "1:"

sets the current time to 01:00:00.

TIME$ = "14:15"

sets the current time to 14:15:00.

TIME$ = "3:3:3"

sets the current time to 03:03:03.

A$ = TIME$

assigns the current time to the variable A$.

# TROFF
# TRON

Turn the "trace function" on/off.

The trace function lets you follow program flow. This is helpful for debugging and analyzing the execution of a program.

Each time the program advances to a new line, TRON displays that line number inside a pair of brackets. TROFF turns the tracer off.

## Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

Lines 2290 and 2310 above might be helpful in assuring you that line 2300 is actually being executed, since each time it is executed [2300] is printed on the display.

After a program is debugged, the TRON and TROFF statements can be removed.

# USR **Function**

## USR[*digit*](*argument*)

Calls a user's assembly-language subroutine identified with *digit* and passes *argument* to that subroutine.

The *digit* you specify must correspond to the *digit* supplied with the DEFUSR statement for that routine. If *digit* is omitted, zero is assumed.

This function lets you call as many as 10 machine-language subroutines, then continue execution of your BASIC program. Subroutines must have been previously defined with DEFUSR[digit] statements.

We recommend that you use the CALL statement to interface assembly language programs with BASIC programs. **Do not** use the USR function unless you are running previously written BASIC programs that already contain USR statements.

Before you can execute a USR function call, you must define the subroutine's address in a DEF SEG and DEF USR statement. The DEF SEG defines the address of the segment containing the subroutine. The DEF USR statement defines the subroutine being called and its offset from the beginning of the segment. This offset and the most recent DEF SEG address specify the entry point of the subroutine. See DEF SEG, DEF USR, and the section "Interfacing Assembly Language Subroutines"in Appendix E.

"Machine language" is the low-level language that your computer uses internally. It consists of 8086 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and for doing things very fast (like to "white-out" the display). Writing such routines requires familiarity with assembly-language programming and with the 8086 instruction set.

# VAL **Function**

## VAL(*string*)

Calculates the numerical value of *string*.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10^34.

VAL terminates its evaluation on the first character which has no meaning in a numeric value.

If the string is non-numeric or null, VAL returns a zero.

## Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 123400000.

```
B = VAL("3" + "*" + "2")
```

assigns the value 3 to B (the asterisk has no meaning in a numeric term).

## Sample Program

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

# VARPTR Function

## VARPTR (*variable*)

## VARPTR (*#buffer*)

Returns the offset into BASIC's data segment of a variable or the file control block.

VARPTR can help you locate a value in memory. When used with *variable*, it returns the address of the first byte of data identified with *variable*. To see the format of how this data is stored see the section "How Variables are Stored" in Appendix E.

When used with *buffer*, it returns the address of the file's file control block.

If the *variable* you specify has not been assigned a value, an "Illegal Function Call" occurs. If you specify a *buffer* that was not allocated when loading BASIC, a "Bad file number" error occurs. (See Chapter 1 for information on how to load BASIC.)

The offset returned is an integer in the range −32768 to 32767. It is always an offset into BASIC's data segment, regardless of whether a DEF SEG has been executed to change the segment.

VARPTR is used primarily to pass a value to a machine-language subroutine via USR[digit]. Since VARPTR returns an offset which indicates where the value of a variable is stored, this address can be passed to a machine-language subroutine as the argument of USR; the subroutine can then extract the contents of the variable with the help of the address that was supplied to it.

If VARPTR returns a negative address, add it to 65536 to obtain the actual address.

# VARPTR$ **Function**

## VARPTR$(*variable*)

Returns a character form of the address of a *variable* in memory.

*variable* is a *variable* name in your BASIC program.

VARPTR$ returns a three byte string in the form:

| | | |
|---|---|---|
| Byte 0 | type | indicates the type of variable. |
| Byte 1 | low order byte of variable address. | |
| Byte 2 | high order byte of variable address. | |

The value returned in Byte 0 is 2 for integer variables, 3 for string variables, 4 for single-precision variables, and 8 for double precision varaibles.

You must assign all simple variables in an array before you use VARPTR$. Addresses of arrays change when you assign a new simple variable.

VARPTR$ is primarily used with the PLAY and DRAW statements in programs that you want to compile because the complier does not support the X subcommand. To execute a substring with compiler, you must append the character form of the address of the substring to "X".

## Example

```
DRAW "XA$;"
DRAW "X" + VARPTR$(A$)
```

These statements are equivalent. The first statement is for interpreter BASIC and the second statement is for compiler BASIC. The second statement appends the address of the variable A$ to the X subcommand.

# WAIT — Statement

## WAIT *port, integer1* [*,integer2*]

Suspends program execution until a machine input *port* develops a specified bit pattern. (A port is an input/output location.)

The data read at the port is exclusive OR'ed with *integer2*, then AND'ed with *integer1*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *integer2* is omitted, it is assumed to be zero.

It is possible to enter an infinite loop with the WAIT statement. In this case, you will have to manually restart the machine. To avoid this, WAIT must have the specified value at port number during some point in program execution.

## Example

```
100 WAIT 32,2
```

# WHILE . . . . WEND — Statement

## WHILE *expression*

•

•

•

•

{loop statements}

•

**WEND**

Execute a series of statements in a loop as long as a given condition is true.

If *expression* is not zero (true), BASIC executes loop statements until it encounters a WEND. BASIC returns to the WHILE statement and checks *expression*. If it is still true, BASIC repeats the process. If it is not true, execution resumes with the statement following the WEND statement.

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

## Sample Program

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

This program sorts the elements in array A$. We assume Array A$ has been defined previously in the program. Control falls out of the WHILE loop when no more SWAPS are performed on line 130.

# WIDTH **Statement**

## WIDTH [LPRINT] *size*
## WIDTH *buffer, size*
## WIDTH *device, size*

Sets the line width in number of characters for the display, line printer, or communication channel.

*size* may be an integer in the range 0 to 255 that specifies the number of characters in a line. For the screen, *size* may only be 40 or 80.

*buffer* is an integer in the range of 0 to 15 and specifies the *buffer* used in the OPEN statement.

*device* is a string expression, enclosed in quotes, that specifies on which *device* you want to set the WIDTH. Valid devices are:

| | |
|---|---|
| SCRN: | indicates the screen. |
| LPT1: or LPT2: | indicates Line Printer 1 or 2. |
| COM1: | indicates the communication channel. |

When you specify a device, BASIC stores the new width and does not change the current width of the device. When a subsequent OPEN statement opens that device, BASIC uses the new width while the file is open. After you close the file, the device returns to the previous width.

When you specify *buffer*, BASIC changes the width immediately. This allows you to change the width when the file is open. To return to the previous width, you must execute another width statement.

When you set the width at the line printer or the communication channel, BASIC sends a carriage return after every *size* characters. If you set the width to 255 for the communication channel, BASIC sends a carriage return after sending the 255th character.

```
10 WIDTH LPRINT 100
20 LPRINT "This line is over 100 characters long.
   See what happens when you print a string longer
   than width setting."
```

Line 10 sets the printer width to 100 characters. After printing 100 characters, BASIC issues a carriage return. The carriage return causes the printer to print the remaining characters on the next line.

To set WIDTH at the screen, you may omit the LPRINT option in the first form of the syntax, like this:

```
WIDTH 40
```

or you may use the third form of the syntax and specify the device, like this:

```
WIDTH "SCRN:", 40
```

You may only use the WIDTH statement to select a WIDTH of 80 if you are using the VM-1 Monochrome Monitor or CM-1 Color Monitor. If you are using the VM-1 Monochrome Monitor or the CM-1 Color Monitor, you should note the following:

1. If you change the screen width, BASIC clears the screen.
2. If you are in Screen Mode 1, changing the WIDTH to 80 forces the screen into Screen Mode 4.
3. If you are in Screen Mode 2 or 4, changing the WIDTH to 40 forces the screen into Screen Mode 1.

If you attempt to select a size outside the range of 0 to 255, an "Illegal function call" error results.

## Examples

```
WIDTH LPRINT 132
WIDTH "LPT1:", 132
```

both of these statements change the printer width to 132. The second statement does not change the printer width until LPT1: is specified as the device in an OPEN statement.

```
10 WIDTH LPRINT 80
.
.
100 OPEN "LPT1:" FOR OUTPUT AS #1
.
150 PRINT #1
.
1000 WIDTH #1, 40
```

Line 10 changes the width of the printer to 80 characters. Line 150 prints the records as 80 characters each. After BASIC executes Line 1000, Line 150 prints the records as 40 characters each.

# WRITE Statement

## WRITE [*data*, . . . ]

Writes *data* on the display.

WRITE prints the values of the data items you type. If *data* is omitted, BASIC prints a blank line. The *data* may be numeric and/or string. They must be separated by commas.

When the *data* is printed, each data item is separated from the last by a comma. Strings are delimited by quotation marks. After printing the last item on the list, BASIC inserts a carriage return. WRITE prints numeric values using the same format as the PRINT statement. See PRINT.

## Example

```
10 D=95:B=76:V$="GOOD BYE"
20 WRITE D, B, V$
RUN
 95, 76, "GOOD BYE"
Ok
```

# WRITE# **Statement**

## WRITE# *buffer, data, . . .*

Writes *data* to a sequential-access file.

*Buffer* must be the number used to OPEN the file.

The *data* you enter may be numeric or string expressions.

WRITE# inserts commas between the data items as they are written to disk. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

The items on *data* must be separated by commas.

WRITE# inserts a carriage return after writing the last data item to disk.

For example, if

```
A$="MICROCOMPUTER" and B$="NEWS"
```

the statement

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

writes the following image to disk:

```
"MICROCOMPUTER","NEWS"
```

Appendices

# Section III / Appendices

# Appendix A

# BASIC Error Codes and Messages

| Code | Number | Message |
|---|---|---|
| | 1 | NEXT without FOR |
| | | A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable. |
| | 2 | Syntax error |
| | | BASIC encountered a line that contains an incorrect sequence of characters (such as unmatched parenthesis, misspelled statement, incorrect punctuation, etc.). BASIC automatically enters the edit mode at the line that caused the error. |
| | 3 | Return without GOSUB |
| | | BASIC encountered a RETURN statement for which there is no matching GOSUB statement. |
| | 4 | Out of data |
| | | BASIC encountered a READ statement, but no DATA statements with unread items remain in the program. |
| | 5 | Illegal function call |
| | | A parameter that is out of range was passed to a math or string function. An FC error may also occur as the result of: |

a. A negative or unreasonably large subscript.

b. A negative or zero argument with LOG.

c. A negative argument to SQR.

d. A negative mantissa with a noninteger exponent.

e. A call to a USR function for which the starting address has not yet been given.

f. An improper argument to MID$, LEFT$, RIGHT$, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON . . . GOTO.

6 Overflow

The result of a calculation was too large to be represented in BASIC numeric format. If underflow occurs, the result is zero and execution continues without an error.

7 Out of memory

A program is too large, or has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 Undefined line number

A nonexistent line was referenced in a GOTO, GOSUB, IF . . . THEN . . . ELSE, or DELETE statement.

9 Subscript out of range

An array element was referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

10 Duplicate defininition

Two DIM statements were given for the same array, or a DIM statement was given for an array after the default dimension of 10 has been established for that array.

11 Division by zero

An expression includes division by zero, or the operation of involution results in zero being raised to a negative power. BASIC supplies machine infinity with the sign of the numerator as the result of the division, or it supplies positive machine infinity as the result of the involution. Execution then continues.

12 Illegal direct

A statement that is illegal in direct mode was entered as a direct mode command.

13 Type mismatch

A string variable name was assigned a numeric value or vice versa. A numeric function was given a string argument or vice versa.

14 Out of string space

String variables have caused BASIC to exceed the amount of free memory remaining. BASIC allocates string space dynamically, until it runs out of memory.

15 String too long

An attempt was made to create a string more than 255 characters long.

16 String formula too complex

A string expression is too long or too complex. The expression should be broken into smaller expressions.

17 Can't continue

An attempt was made to continue a program that:

a. Has halted due to an error.

|  |  |
|---|---|
|  | b. Has been modified during a break in execution. |
|  | c. Does not exist. |
| 18 | Undefined user function. |
|  | A USR function was called before providing a function definition (DEF statement). |
| 19 | No RESUME |
|  | An error-handling routine was entered without a matching RESUME statement. |
| 20 | RESUME without error |
|  | A RESUME statement was encountered prior to an error-handling routine. |
| 21 | Unprintable error |
|  | An error message is not available for the error that occurred. |
| 22 | Missing operand |
|  | An expression contains an operator with no operand. |
| 23 | Line buffer overflow |
|  | An attempt was made to input a line with too many characters. |
| 24 | Device Timeout |
|  | BASIC did not receive information from an I/O device within a predetermined amount of time. |
| 25 | Device Fault |
|  | Indicates a hardware error in the printer or interface card. |
| 26 | FOR without NEXT |
|  | A FOR statement was encountered without a matching NEXT. |

27 Out of paper

The printer is out of paper.

29 WHILE without WEND

A WHILE statement does not have a matching WEND.

30 WEND without WHILE

A WEND statement was encountered without a matching WHILE.

**Disk Errors**

50 Field overflow

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a direct-access file.

51 Internal error

An internal malfunction has occurred in BASIC. Report the conditions under which the message appeared to Radio Shack.

52 Bad file number

A statement or command references a file with a buffer number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found

A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode

An attempt was made to use PUT, GET, or LOF with a sequential file, to LOAD a direct file, or to execute an OPEN statement with a file mode other than I, O, R, or A.

| | |
|---|---|
| 55 | File already open |
| | An OPEN statement for sequential output was issued for a file that is already open; or a KILL statement was given for a file that is open. |
| 57 | Device I/O error |
| | An Input/Output error occurred. This is a fatal error; the operating system cannot recover it. |
| 58 | File already exists |
| | The filespec specified in a NAME statement is identical to a filespec already in use on the disk. |
| 61 | Disk full |
| | All disk storage space is in use. |
| 62 | Input past end |
| | An INPUT statement was executed after all the data in the file had been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file. |
| 63 | Bad record number |
| | In a PUT or GET statement, the record number is either greater than the maximum allowed (32,767) or equal to zero. |
| 64 | Bad file name |
| | An illegal filespec (file name) was used with a LOAD, SAVE, KILL, or OPEN statement (for example, a filespec with too many characters). |
| 66 | Direct statement in file |
| | A direct statement was encountered while LOADing an ASCII-format file. The LOAD is terminated. |

67 Too many files

An attempt was made to create a new file (using SAVE or OPEN) when all directory entries are full.

68 Device Unavailable

An attempt was made to open a file to a nonexistent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:, or was disabled by the user. This occurs if an OPEN "COM1: . . . statement is executed but the user disabled RS232 support via the /C:0 switch directive on the command line.

69 Communication buffer overflow

Occurs when a communication input statement is executed and the input queue is already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case several options are available:

a. Increase the size of the COM receive buffer via the /C: switch.

b. Implement a "hand-shaking" protocol with the host/satellite such as XON/XOFF to turn transmit off long enough to catch up.

c. Use a lower Baud rate for transmit and receive.

70 Disk Write Protect

This is one of 3 "hard" disk errors returned from the diskette controller. This occurs when an attempt is made to write to a diskette that is write protected. Use an ON ERROR GOTO statement to detect this situation and request operator action.

71 Disk not Ready

Occurs when the diskette drive door is open or a diskette is not in the drive. Again use an ON ERROR GOTO statement to recover.

72 Disk Media Error

Occurs when the FDC controller detects a hardware or media fault. This usually indicates harmed media. Copy any existing files to a new diskette and re-format the damaged diskette. FORMAT flags the bad tracks and places them in a file "badtrack". The remainder of the diskette is now usable.

74 Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed.

# Appendix B

# BASIC Reserved Words and Derived Functions

## Reserved BASIC Words

- ABS
- AND
- ASC
- ATN
- AUTO
- BEEP
- BLOAD
- BSAVE
- CALL
- CDBL
- CHAIN
- CHR$
- CINT
- CIRCLE
- CLEAR
- CLOSE
- CLS
- COLOR
- COM
- COMMON
- CONT
- COS
- CSRLIN
- CSNG
- CVD
- CVI
- CVS
- DATA
- DATE$
- DEF
- DEFDBL
- DEFINT
- DEFSNG
- DEFSTR
- DEF FN
- DEF USR
- DELETE
- DIM
- DRAW
- EDIT
- ELSE
- END
- EOF
- ERASE
- ERL
- ERR
- ERROR
- EXP
- FIELD
- FILES
- FIX
- FN
- FOR
- FRE
- GET
- GOSUB
- GOTO
- HEX$
- IF
- IMP
- INKEY$
- INP
- INPUT
- INPUT#
- INPUT$
- INSTR
- INT
- KEY
- KILL
- LEFT$
- LEN
- LET
- LINE
- LIST
- LLIST
- LOAD
- LOC
- LOCATE
- LOF
- LOG
- LPOS
- LPRINT
- LSET
- MERGE
- MID$
- MKD$
- MKI$
- MKS$
- MOD
- MOTOR
- NAME
- NEW
- NEXT
- NOT
- OCT$
- OFF
- ON
- OPEN
- OPTION
- OR
- OUT
- PAINT
- PALETTE
- PALETTE USING
- PEEK
- PEN
- PLAY
- POINT
- POKE
- POS
- PRESET
- PRINT
- PRINT#
- PSET
- PUT
- RANDOMIZE
- READ
- REM
- RENUM
- RESET
- RESTORE
- RESUME
- RETURN
- RIGHT$
- RND
- RSET
- RUN
- SAVE
- SBN
- SCREEN
- SGN
- SIN
- SOUND
- SPACE$
- SPC
- SQR
- STEP
- STICK
- STOP
- STR$
- STRIG
- STRING$
- SWAP
- SYSTEM
- TAB
- TAN
- THEN
- TIME$
- TO
- TROFF
- TRON
- USING
- USR
- VAL
- VARPTR
- VARPTR$
- WAIT
- WEND
- WHILE
- WIDTH
- WRITE
- WRITE#
- XOR

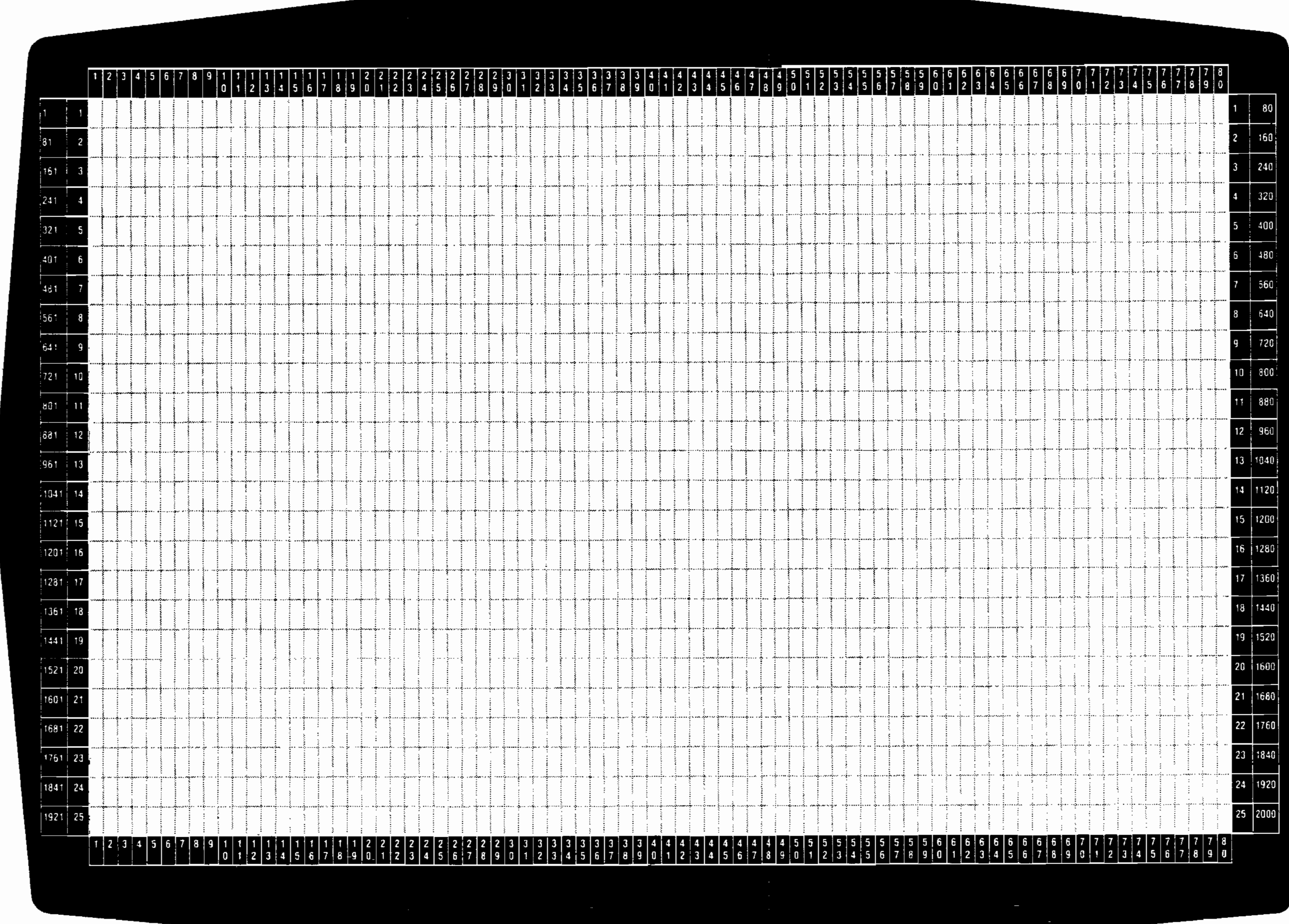
# Derived BASIC Functions

Functions which are not intrinsic to BASIC may be calculated as follows:

| Function | BASIC 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 | ARSCEC(X) = ATN(X/SQR(X*X − 1)) + (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 |

# Appendix C Video Display Worksheet



# Appendix D

# Memory Map

| Hexadecimal Address (Segment:Offset) | Description |
|---|---|
| 0000:0 to 0AFF:F | System Area |
| 0B00:0 to xA00:0 | Available to User<br>See Note on next page |
| xB00:0 to xB39:0 | Hardware Stack<br>512 levels |
| xB40:0 to xCFF:0 | Video Display RAM<br>Screen Buffers<br>Room for eight 40 x 25 display pages or four 80 x 25 display pages |
| C000:0 to DFFF:0 | Not used — Reserved |
| E000:0 to E7FF:0 | Graphics RAM<br>Low- or<br>High-Resolution<br>Graphic Boards |
| E800:0 to F7FF:0 | Not used — Reserved |
| F800:0 to F9FF:0 | Video Character RAM |
| FA00:0 to FBFF:0 | Not used — Reserved |
| FC00:0 to FFFF:0 | Boot ROM |

Note: Additional memory must be added in 128 K byte increments. This key can be used to determine the value of x in the above addresses. Memory Size is the total amount of memory you have in your system. n is the number you additional 128 K bytes of memory that have been added in addition to the standard system. If you have not added any additional memory to your system, in the memory map above x is equal to 1.

| Memory Size | n Value | x Value |
|---|---|---|
| 128 K | 0 | 1 |
| 256 K | 1 | 2 |
| 384 K | 2 | 3 |
| 512 K | 3 | 4 |
| 640 K | 4 | 5 |
| 768 K | 5 | 6 |

# Appendix E

# Technical Information

## Interfacing with Assembly Language Subroutines

This section is for users who call subroutines written in languages other than BASIC, from their BASIC programs. BASIC provides for interfacing with subroutines through the USR function and through the CALL and the CALLS statements.

The USR function allows you to call assembly language subroutines in the same way BASIC calls intrinsic functions. However, we recommend CALL or CALLS statements for interfacing 8086 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 is the USR function.

## Memory Allocation

You can load your assembly language subroutine into BASIC's work area or into another segment of memory. We show you both methods.

### Outside the BASIC work area

When you load BASIC, the DS (data segment) register is set to the address of BASIC's workarea. To access an area of memory outside this workarea, you must execute a DEF SEG statement to specify the address of the segment of memory you are accessing. If you don't execute a DEF SEG statement, your CALL, CALLS, or USR statements transfer control to an area within BASIC's workarea. After returning from the subroutine, you must execute another DEF SEG statement to restore the DS register to its original value. See "Chapter 7 BASIC Keywords" DEF SEG statement for more information on DEF SEG.

### Inside the BASIC work area

To set aside memory space for an assembly language subroutine within BASIC's workarea, use the /M: switch when you load BASIC. See Chapter 1 for a review of the start-up procedure.

The /M: switch sets the highest memory address that BASIC can use. The value that you specify with the /M: switch tells BASIC that it can use all memory up to that offset. Load your subroutine at that offset. Using the /M: switch will prevent BASIC from destroying your subroutine. For example,

```
BASIC /M:&HF000
```

sets the highest memory location that BASIC can use at hexadecimal address EFFF. This reserves the highest 4K bytes of memory for your subroutine. You can load your subroutine at hexadecimal address &HF000 like this:

```
BLOAD "SUBA.ASM",&HF000
```

### Stack Space

If you need more stack space when you call an assembly language subroutine, you can save the BASIC stack and set up a new stack for the subroutine. You must restore the BASIC stack before returning from the subroutine. You save the stack, create a new stack, and restore the stack in your subroutine.

## Loading the Subroutine into Memory

You can use the operating system or the POKE statement to load the subroutine into memory. You may assemble the routines with the Macro Assembler (Catalog Number 26-5252), and link (but not load) them with Linker. The Linker is part of the MS-DOS package. To load the program file, observe these guidelines:

1. Be sure that the subroutines do not contain any long references.

2. Skip the first 512 bytes of the LINK output file, and then read in the rest of the file.

### Poking a Subroutine into Memory

You can code short subroutines in machine language and use the POKE statement to put the code into memory. To do so, follow these steps:

1. Code the machine language instructions for your subroutine.

2. Put the assembly opcode for each byte of the machine language code into DATA statements, preceded by the &H symbols to denote that they are hexadecimal values.

3. Execute a loop that reads the DATA statements and POKEs them into an area of memory.

For example, the opcode for the statement

```
PUSH    BP
```

is 55. The DATA statement for that instruction is

```
DATA    &H55
```

After the loop is complete, the subroutine is in memory. Whether you are using the USR function or the CALL statement to call the subroutine, you must set the value of the subroutine entry point as the location specified in the first POKE statement.

## CALL Statement

We recommend that you use the CALL statement to interface 8086 machine language programs with BASIC. Do not use the USR function unless you are running previously written programs that already contain USR functions.

### CALL *variable* [*parameter list*]

*variable* is a variable in your BASIC program that contains the offset into a segment where the subroutine starts.

*parameter list* contains the variables or constants, separated by commas, that are passed to the subroutine.

The number, length, and type (string, integer, single precision, or double precision) of variables passed in the CALL statement must match the number, length, and type of variables expected by the subroutine.

Example:

```
100 MYROUT = &H0000
110 DEF SEG = &H1700
120 CALL MYROUT, HRS!, RATE!, PAY!
```

Line 100 defines the subroutine address at offset 0. Line 110 defines the segment address of the subroutine. Line 120 transfers program control to the subroutine, passing it the variables HRS!, RATE!, and PAY!. In this example, HRS! and RATE! are variables for the subroutine to perform the calculation of weekly pay. When the subroutine returns program control to BASIC, PAY! contains the result of the calculation.

## Entry Conditions

When the CALL statement is executed, the following occur:

1. For each parameter in the parameter list, the two-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack. If the parameter is a string variable, the offset points to the *string descriptor*. See the section "Accessing String Parameters" in this appendix.

2. The BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.

3. Control is transferred to the subroutine by an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in *variable*.

This diagram illustrates the state of the stack when the CALL is executed.

## Subroutine Address

When the CALL statement is executed, the operating system loads the CS (code segment) register with the value specified in the last DEF SEG statement. If you are CALLing

a subroutine within BASIC's workarea, and no DEF SEG is required, the CS register is loaded with the address of BASIC's workarea. This address is shifted left four bits; in other words, a zero is appended like this: 17000. Then the offset of the subroutine is added to the segment address.

Example:

```
17000 + 0020 = 17020
```

17020 is the absolute address of the first instruction in the subroutine.

## Technical Functions

The CALLed routine may destroy the previous contents of all registers. If you want to save the contents of the registers, the first instructions in the subroutine must be a PUSH for each register and the last instructions in the subroutine must be a POP to restore the registers to their original value. You must execute a POP for every PUSH to maintain stack integrity.

The subroutine may refer to the passed parameters as positive offsets to the Base Pointer (BP). The CALLed routine must PUSH BP on the stack and then move the current stack pointer into BP. BP should be the first register you PUSH so that the parameters may be referenced as an offset to BP. The first four bytes of the stack contain the IP and CS register values that BASIC saves when the CALL is executed. To calculate the parameters offset from the BP, use this equation:

2 * (total parameters – parameter position)
 + 6 = offset

For example, the address of parameter 1 is at 10(BP), parameter 2 is at 8(BP), and parameter 3 is at 6(BP).

**Example**

```
PUSH    BP          ;save BP
MOV     BP,SP       ;current stack position in BP
MOV     BX,10[BP]   ;get address of HRS! dope
```

### Exit Conditions

The called routine must execute a RET *number* statement to adjust the stack to the start of the calling sequence. The value of *number* is two times the number of parameters in the parameter list.

**Example**

```
RET    6
```

*number* is 6 for our sample because three parameters were passed.

## USR Function

Although we recommend the CALL statement for calling assembly language subroutines, the USR function is available for compatibility with previously written programs.

### USR [*digit*] (*argument*)

*digit* is in the range 0 to 9. *digit* specifies which USR routine is being called and must correspond to the *digit* supplied in the DEFUSR statement. If you omit *digit*, BASIC assumes USR0.

*argument* is any numeric or string expression. Even if the function that is called does not need an argument, you must supply a dummy argument.

Example

```
100 DEF USR2 = &H0020
110 DEF SEG = &H1700
120 E = USR2(A)
```

Line 100 defines the USR2 subroutine's address at offset hexadecimal 20. Line 110 defines the segment address of the subroutine. Line 120 transfers program control to the subroutine, passing it parameter A. When the subroutine returns program control to BASIC, E contains the result of the subroutine calculations.

USR can only pass one value to a subroutine, and it can only receive one value from the subroutine after execution. The value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

## Entry Conditions

When the USR statement is executed, the operating system loads the CS (code segment) register with the value specified in the last DEF SEG statement. If you are accessing a subroutine within BASIC's work area and no DEF SEG is required, the CS register is loaded with the address of BASIC's workarea. This address is shifted left four bits; in other words, a zero is appended like this: 17000. Then the offset of the subroutine is added to the segment address.

Example:

```
17000 + 0020 = 17020
```

17020 is the absolute address of the first instruction in the subroutine.

## Technical Functions

When the subroutine gains control, register AL contains a value that specifies the type of argument that 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 string, the DX register pair points to the "string descriptor." See the section "Accessing String Variables" in this appendix.

If the argument is a number, the BX register pair points to the Floating-Point Accumulator (FAC) where the argument is stored:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative).

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 mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double-precision floating-point number:

FAC-4 through FAC-7 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

### Exit Conditions

The subroutine must execute a RET 2 statement to adjust the stack to the start of the calling sequence.

## Accessing String Variables

If the parameter passed in a CALL statement is a string expression, the parameters offset points to the *string descriptor*. If the argument passed in a USR function call is a string expression, the DX register points to the *string descriptor*.

The *string descriptor* is a three-byte area of memory that points to the text of the string. The *string descriptor* contains the following:

Byte 0 contains the length of the string (0 to 255).

Byte 1 contains the lower eight bits of the string start address in BASIC's data segment.

Byte 2 contains the upper eight bits of the string start address in BASIC's data segment.

The text of the string may be altered by the subroutine, but the length of the string **must not** be changed. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Since the *string descriptor* points to an area of memory in your BASIC program, you must be careful not to alter or destroy your program. To avoid unpredictable results, add the concatenation symbol ( + ) to the string. This forces the string to be copied into string space, where the string may be modified without affecting the program.

Example

```
20 A$ = "MONTHLY SALES REPORT" + ""
```

# File Control Block

A file control block is a storage area in BASIC's data segment that contains information BASIC needs for all functions performed on that file. When you execute the VARPTR function and specify the buffer number, BASIC returns the address of the BASIC file control block for that file. Note that this is the BASIC file control block, not the MS-DOS file control block. The address is specified as an offset into BASIC data segment. In this section we define the information in the file control block. Offsets are relative to the value returned by VARPTR. Length is in bytes.

| OFFSET | LENGTH | DESCRIPTION | |
|---|---|---|---|
| Ø | 1 | Mode | The mode in which the file was opened:<br>1 — Input Only<br>2 — Output Only<br>4 — Random I/O<br>16 — Append Only<br>32 — Internal use<br>64 — Future use<br>128 — Internal use |
| 1 | 38 | FCB | MS-DOS Disk File Control Block. |
| 39 | 2 | CURLOC | Number of sectors read or written for sequential access. For random access, it contains the last record number +1 read or written. |
| 41 | 1 | ORNOFS | Number of bytes in sector when read or written. |
| 42 | 1 | NMLOFS | Number of bytes left in Input buffer. |

| | | | |
|---|---|---|---|
| 43 | 3 | *** | Reserved for future expansion. |
| 46 | 1 | DEVICE | Device number:<br>0-4 — Disks A: thru D:<br>249 — LPT2:<br>251 — COM1:<br>253 — LPT1:<br>254 — SCRN:<br>255 — KYBD: |
| 47 | 1 | WIDTH | Device width. |
| 48 | 1 | POS | Position in buffer for PRINT. |
| 49 | 1 | FLAGS | Internal use during LOAD/SAVE; not used for data files. |
| 50 | 1 | OUTPOS | Output position used during tab expansion. |
| 51 | 128 | BUFFER | Physical data buffer. Used to transfer data between MS-DOS and BASIC. Use this offset to examine data in Sequential I/O mode. |
| 179 | 2 | VRECL | Variable length record size. Default is 128. Set by length option in OPEN statement. |
| 181 | 2 | PHYREC | Current physical record number. |
| 183 | 2 | LOGREC | Current logical record number. |
| 185 | 1 | *** | Future use. |

| | | | |
|---|---|---|---|
| 186 | 2 | OUTPOS | Disk files only. Output position for PRINT, INPUT, and WRITE. |
| 188 | <n> | FIELD | Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine File data in Random I/O mode. |

# How Variables are Stored

BASIC stores variables in its data segment as follows:

| **Byte** | **Contents** | **Description** |
|---|---|---|
| Byte 0 | Type | Identifies the type of variable stored at this location: |
| | 2 | integer |
| | 3 | string |
| | 4 | single-precision |
| | 5 | double-precision |
| Bytes 1 and 2 | Name | The first two characters of the variable name. |
| Byte 3 | Integer 3 - 38 | Integer is the number of additional characters in the variable name. |
| Byte 4 + integer stored in byte 3 | Name | The remainder of the variable name is stored at bytes 4 + the integer stored in byte 3. |

| | |
|---|---|
| Byte 4 + length Data length | The contents of the variable are stored in the bytes immediately following the variable name. The data can be two, three, four, or eight bytes in length, depending on the type of data. |

At least three bytes are required to store any variable name. A one- or two-character variable name occupies exactly three bytes, bytes one and two for the first two characters and byte three contains a zero to indicate that there are no additional characters in the variable name. If the variable name only contains one or two characters, the data is stored beginning at byte four. As you can see, the location of the first actual byte of data depends on the length of the variable name. VARPTR returns the offset of the first actual byte of data, not the offset of the beginning of the storage area.

# Index


A

Absolute value of number . . . . 94
ABS . . . . 94
Active Page . . . . 291
Addition . . . . 55
Address, Character form of. . . . 322
ALL option . . . . 104, 126
ALT Key for BASIC Keyword Entry . . . . 19
AND . . . . 59, 270-271
Animation. . . . 166, 270-271
Apostrophe for a remark . . . . 277
Arctangent, Computing the . . . . 96
Arrays . . . . 40-43, 143, 152, 166, 236
ASCII
  Code of character on screen . . . . 290
  Codes, printing . . . . 311
  Compare . . . . 56
  Converting string to (ASC) . . . . 95
  Converting to string (CHR$) . . . . 108
  Format, Saving files in . . . . 104, 288-289
Aspect Ratio . . . . 89-92, 111
Assembly language subroutines . . . . 102, 115, 141, 252, 319, 321, 347-354
ATN . . . . 96
AUTO . . . . 97
Automatic Keyword Entry . . . . 19
Automatic Line Number Entry . . . . 97

B

Background colors. . . . 117, 118, 121
BEEP . . . . 98
Binary File, encoded . . . . 288
Bits per Point. . . . 166
Blanks . . . . 298-299
BLOAD . . . . 99
Boolean Operators . . . . 59
Border colors. . . . 118
Branch
  Conditional. . . . 170, 172, 173, 223-225
  on an error . . . . 223
  to a line number . . . . 170, 225
  to a subroutine . . . . 169, 283
Buffer
  File . . . . 2, 65, 71, 157, 178, 180, 205, 211, 230, 268, 286, 321
  Screen . . . . 117, 291

C

/C: . . . . 10, 203
CALL . . . . 102, 319, 349-352
CDBL . . . . 103
CHAIN . . . . 104, 126
Chaining Programs . . . . 104, 126
CINT. . . . 109
CIRCLE. . . . 110
Clearing
  the Screen . . . . 117
  Memory . . . . 218, 287
Clock Ticks . . . . 296-297
CLOSE . . . . 116
Closing files. . . . 115, 149, 280, 287
CLS . . . . 117
COLOR. . . . 116-123
Color . . . . 241-245, 265-267
  of a point on the screen . . . . 251, 290
  To enable and disable . . . . 291
Column, Screen. . . . 204, 253
COM(1) ON, OFF, STOP . . . . 124, 151, 220
Command Mode
  prompt. . . . 15
  special keys in. . . . 17-19
Comments in a program. . . . 277
COMMON. . . . 104, 126
Communication
  Channel . . . . 124, 232-234, 325
  Files
    EOF . . . . 151
    Number of characters in queue. . 203, 207
    Transferring data . . . . 166, 269
    Transmitting data. . . . 232-234
    Trapping. . . . 124, 220
Compressed Files . . . . 288-289
Concatenation . . . . 55
Constants. . . . 38, 46
CONT . . . . 127, 303
Continue program execution . . . . 127
Control Key functions in
  Command Mode . . . . 17-19
  Edit Mode . . . . 25-27
  Execution Mode. . . . 19
Converting
  ASCII to character. . . . 108
  numeric data . . . . 47-50, 103, 109, 184, 131
  numeric to string. . . . 71, 216
  string to ASCII. . . . 95
  string to numeric. . . . 73, 132
Coordinate, Movement on. . . . 301
COS . . . . 129
Cosine of number . . . . 129

CSNG . . . . 132
CSRLIN . . . . 130
Current date. . . . 135
Current time. . . . 316
Cursor position . . . . 117, 130, 204, 253
CVD$, CVI$, CVS$ . . . . 73, 132

D

DATA . . . . 133, 182, 276, 281
DATE$ . . . . 135
Dates, Valid . . . . 135
Debugging . . . . 127, 303, 318
Defining variables . . . . 137
Defining functions . . . . 138
Defining USR subroutine. . . . 141
DEF FN. . . . 107, 138
DEF USR . . . . 141, 319
DEFDBL . . . . 46, 107, 137
DEFINT. . . . 46, 107, 137
Definition statements. . . . 46, 47, 137
DEFSNG . . . . 46, 107, 137
DEF SEG . . . . 99, 101, 102, 140, 319, 347
DEFSTR . . . . 46, 107, 137
Degrees
  cosine of . . . . 129
  sine of . . . . 295
  tangent of . . . . 315
  to convert radians to. . . . 96
DELETE . . . . 76, 142
Deleting
  a program . . . . 218, 287
  files from a disk . . . . 190
  program lines . . . . 104, 142
Dimensioning an array . . . . 43
Directory . . . . 158, 280, 313
Direct Access. . . . 70-74
  Creating . . . . 70-72, 268
  Updating and Accessing . . . . 72-74
Division
  Ordinary. . . . 52,53
  Integer . . . . 52,54
Documentation. . . . 277
Double Precision . . . . 37
  constants . . . . 46
  converting . . . . 48-50, 103
DRAW . . . . 145
Draw a point on the screen. . . . 265-267
Drive Identifier. . . . 12

E

Edit Mode . . . . 21-24
  Sample Session. . . . 25-27
  Special keys in. . . . 25-27
EDIT Statement . . . . 21, 148
Editing output . . . . 256-260
Element of an array. . . . 40, 143
Ellipse . . . . 110-112
Encoded Binary File. . . . 288-289
END . . . . 149
EOF (End Of File)
  Disk File. . . . 150
  Communication File. . . . 151
ERASE . . . . 152
ERL . . . . 153, 287
ERR . . . . 154
ERROR. . . . 153-155
Error
  Code. . . . 154
  Handling routine . . . . 153-155, 223, 282
  Line number of . . . . 153
  Messages. . . . 333-340
  to simulate. . . . 155
Exchanging values of variables . . . . 312
Executing a program. . . . 287
Executing MS-DOS Commands. . . . 313
Execution Mode. . . . 15
  Special keys in. . . . 19
EXP . . . . 156
Exponent . . . . 156
Exponential Format . . . . 35
Exponentiation. . . . 52,53
Expression . . . . 33
Extension . . . . 12, 288

F

/F:. . . . 10
FIELD statement . . . . 71-74, 157
File
  buffer . . . . 65, 116
  closing . . . . 115, 116
  Control Block. . . . 321, 356
  direct . . . . 70-74
  sequential. . . . 65-69
File protection . . . . 288-289
FILES . . . . 158
Filespec . . . . 12
FIX . . . . 159
Fixed point
  constants . . . . 38

numerics .......... 35
Floating Point
constants .......... 38
numerics .......... 35
FOR/NEXT .......... 160-162
Foreground colors .......... 117, 118, 121
Formatting data .......... 256-260
FRE .......... 163
Function .......... 34
Brief definition of each .......... 81-83
Defining .......... 138
Keys .......... 185-189

G

GET (Communication Files) .......... 165
GET (Disk files) .......... 73, 132, 164
GET (Graphics) .......... 166
GOSUB .......... 169, 283
Graphic Images .......... 145, 166
Graphics Options
Colors .......... 118, 121
Coordinates .......... 85, 87, 88, 89
Modes .......... 85-91, 291

H

Hexadecimal
Constants .......... 38
Numerics .......... 36
Value of a number .......... 171

I

IF/THEN/ELSE .......... 57, 172, 173
Immediate Lines .......... 16
INKEY$ .......... 174
INPUT .......... 66, 72
Input from keyboard .......... 175, 176, 180, 197
Input queue .......... 151, 203
INPUT# .......... 66,68, 178, 263
INPUT$ .......... 180
INSTR .......... 182
INT .......... 184
Integer .......... 2
constants .......... 38,46
convert number to .......... 109, 159, 184
converting .......... 48-50
defining (DEFINT) .......... 46
division .......... 52, 53
numerics .......... 35-36
Invisible characters .......... 116, 122, 265-267

K

KEY
OFF .......... 185-189
ON .......... 185-189
LIST .......... 185-189
Set/Display .......... 185-186
STOP .......... 185-189
Trap .......... 187-189, 226
( ) ON .......... 226
( ) OFF .......... 226
( ) STOP .......... 226
Keyboard
As an input device .......... 230
Input .......... 174, 176, 197
KILL .......... 190

L

Last Record in a file .......... 150
Left Justify .......... 211
LEFT$ .......... 191
LEN .......... 192
LET .......... 193
LINE .......... 194-196
Line,
Drawing on the screen .......... 194-196
Immediate .......... 16
Logical .......... 15
Numbers .......... 2, 31, 278-279
Numbers, Automatic Entry .......... 97
Physical .......... 15
Program .......... 16
Line Editor .......... 21
Line Feed .......... 289
LINE INPUT .......... 197
LINE INPUT# .......... 66-68, 198
LIST .......... 199
Listing to printer .......... 199, 200, 210
Listing to screen .......... 199
LLIST .......... 200
LOAD .......... 201
Loading
a memory image file .......... 99
a program .......... 14, 201
assembly language subroutines .......... 347-349
BASIC .......... 9
Graphics BASIC .......... 9

LOC (Communication Files) .......... 203
LOC (Disk Files) .......... 202
LOCATE .......... 204
LOF (Communication Files) .......... 207
LOF (Disk Files) .......... 73, 74, 205
LOG .......... 208
Logarithm .......... 208
Logical
  Expression .......... 33
  Lines .......... 15
  Operators .......... 58
Loops .......... 161, 162, 324
LPOS .......... 209
LPRINT .......... 210, 314
LPRINT USING .......... 210
LSET .......... 71,72, 211

M

/M: .......... 10, 99, 115, 140, 348
Memory
  Address .......... 238
  Available .......... 163
  Clearing .......... 115, 218
  Map .......... 345
  Writing data in .......... 252
Memory image file .......... 99, 101
MERGE
  Option .......... 104, 288-289
  Statement .......... 212-213
Merging programs .......... 104, 212-213
MKD$, MKI$, MKS$ .......... 71
Modes, Graphics and text .......... 291
Modulus Arithmetic .......... 52, 54
Mouse button trapping .......... 228, 306-307, 308-310
MS-DOS command level prompt .......... 9, 313
MS-DOS commands, executing .......... 313
Multiplication .......... 52, 54
Musical notes .......... 248, 296-297

N

NAME .......... 217
Naming files .......... 217
  Rules for .......... 13
Nested loops .......... 161, 162, 172-173
Numeric
  Constants .......... 38
  Conversions .......... 48-50, 71, 73, 103, 132, 216
  Expression .......... 33
  Operators .......... 52-54
  Precision .......... 36
  Relations .......... 56
  Types of .......... 35
Number
  Absolute value of .......... 94
  Arctangent of .......... 96
  Converting .......... 48-50, 71, 73, 103, 109, 132, 159, 184, 216, 304, 320
  Cosine of .......... 129
  Exponent of .......... 156
  Hexadecimal value of .......... 171
  Logarithm of .......... 208
  Octal value of .......... 219
  Sign of .......... 294
  Sine of .......... 295
  Square root of .......... 300
  Tangent of .......... 315

O

Octal
  Constants .......... 38
  Numerics .......... 36
  Value of a number .......... 219
OCT$ .......... 219
Offset, Address .......... 99, 101, 102, 141, 319, 321
Offset, Coordinate .......... 110
ON COM(1) .......... 124, 220
ON ERROR GOTO .......... 154, 155, 223
ON GOSUB .......... 224
ON GOTO .......... 225
ON KEY .......... 187-189, 226
ON STRIG .......... 228, 306-307
OPEN (Disk Files) .......... 66-70, 72, 74, 230-231
OPEN COM1 (Communication Files) .......... 232-234, 269
Operands .......... 51
Operators
  Logical .......... 58
  Numeric .......... 52-54
  Relational .......... 56
  String .......... 55
OPTION BASE .......... 106, 143, 236
OR .......... 59, 270-271
OUT .......... 237

## P

Page, active and visual . . . . . 291
PAINT . . . . . 239
Palette . . . . . 121, 241
PALETTE . . . . . 241
PALETTE USING . . . . . 245
Parameter, passing to a subroutine . . . 102, 252
Passing variables . . . . . 104, 126
PEEK . . . . . 238
Physical Lines . . . . . 15
PLAY . . . . . 248
POINT . . . . . 251
POKE . . . . . 252, 349
Ports . . . . . 237, 324
POS . . . . . 253
PRESET . . . . . 251, 265-267, 270-271
PRINT . . . . . 73, 254-255, 314
PRINT # . . . . . 178, 262
Printer
  listing . . . . . 199, 200, 210
  output to . . . . . 230, 314
  position of head . . . . . 209
  width of . . . . . 325
PRINT TAB . . . . . 261, 314
PRINT USING . . . . . 74, 256-260
Program . . . . . 31, 76
  ending . . . . . 149
  deleting from memory . . . . . 218
  loading into memory . . . . . 201
  loops . . . . . 161, 162, 172, 173, 283, 324
  merging . . . . . 104
  overlaying . . . . . 104
  pause exection . . . . . 303
  running . . . . . 201
Program Lines . . . . . 16
Program Listing
  to line printer . . . . . 199, 200
  to screen . . . . . 199
Prompt for keyboard input . . . . . 176, 197
Protection, file . . . . . 288-289
PSET . . . . . 251, 265-267, 270-271
PUT (Communication Files) . . . . . 269
PUT (Disk Files) . . . . . 71, 268
PUT (Graphics) . . . . . 270-271

## R

Radians
  computing . . . . . 96
  cosine of . . . . . 129
  sine of . . . . . 295
  tangent of . . . . . 315
Radius of a circle . . . . . 110
RANDOM mode . . . . . 70, 230
Random numbers . . . . . 275, 285
RANDOMIZE . . . . . 275
Ratio, aspect . . . . . 89-92, 111
READ . . . . . 133, 276, 281
Reading data . . . . . 133, 178, 276, 281
Real Numbers . . . . . 35
Records
  current number . . . . . 202
  detecting last in a disk file . . . . . 150
  length . . . . . 230
  maximum number . . . . . 70
  number . . . . . 71-72, 164, 202
  reading from disk . . . . . 164
  size and default size . . . . . 70
  writing to disk . . . . . 268
Relational
  expression . . . . . 33
  how to use relational expressions . . . . . 57
  operators . . . . . 56
REM . . . . . 277
Remarks in a program . . . . . 277
RENUM . . . . . 278-279
Renaming Files . . . . . 217
Reserved Words . . . . . 341
RESET . . . . . 280
RESTORE . . . . . 133, 276, 281
RESUME . . . . . 223, 282
RETURN . . . . . 169, 283
Return to MS-DOS . . . . . 313
RIGHT$ . . . . . 284
Right portion of a string . . . . . 284
RND . . . . . 275, 285
Row, screen . . . . . 130, 204
RSET . . . . . 286
RUN . . . . . 14
Running a program . . . . . 14, 201

## S

/S: . . . . . 10, 230
SAVE . . . . . 288-289
Saving
  a memory image file . . . . . 101
  a program . . . . . 12, 288-289
  a program in ASCII format . . . . . 104
SCREEN (Statement) . . . . . 117
SCREEN (Function) . . . . . 290
Screen
  buffer . . . . . 117
  clearing . . . . . 117

images . . . . . . . . . . . . . . . . . . . . . . . 270-271
modes, selecting . . . . . . . . . . . . . . . . . . . 291
printing data to . . . . . . . . . . . . . . . . 254-261
program listing . . . . . . . . . . . . . . . . 199, 200
row. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
spacing . . . . . . . . . . . . . . . . . . . . . . . . . 314
width . . . . . . . . . . . . . . . . . . . . . . . 291, 325
Segment. . . . . . . . . . . . . . . . . . . . . . . . 140, 141
Sequential Files . . . . . . . . . . . . . . . . . . . . 65-69
Accessing and Updating . . . . . . . 68-69, 178, 198, 230-231, 329
Creating . . . . . . . . .66-67, 230-231, 262, 329
Setting the time . . . . . . . . . . . . . . . . . . . . . 316
SGN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
SIN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295
Sine of a number. . . . . . . . . . . . . . . . . . . . 295
Single Precision . . . . . . . . . . . . . . . . . . . . . . 37
constants . . . . . . . . . . . . . . . . . . . . . . . . . 46
converting . . . . . . . . . . . . . . . . . . . . . 48-50
defining (DEFSNG). . . . . . . . . . . . . . . . . . . 46
Soft keys . . . . . . . . . . . . . . . . . . . . . . 185-189
SOUND. . . . . . . . . . . . . . . . . . . . . . . . 296-297
Sounds, to generate . . . . . . . .98, 248, 296-297
Spaces . . . . . . . . . . . . . . . . . . . . . . . . . . . 298
SPACE$ . . . . . . . . . . . . . . . . . . . . . . . . . . 298
SPC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 299
SQR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300
Square Root of Number . . . . . . . . . . . . . . . 300
Stack Space . . . . . . . . . . . . . . . . . . . . . 115,348
Statement. . . . . . . . . . . . . . . . . . . . . . . . . . . 31
brief definition of each . . . . . . . . . . . . . 76-81
Status of mouse buttons. . . . . . . . . . . . 308-310
STICK. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 301
STOP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 303
STR$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 304
STRIG (Function). . . . . . . . . . . . . . . . . . . . 305
STRIG (Statement) . . . . . . . . . . . . . . . 308-310
STRIG OFF. . . . . . . . . . . . . . . . . . . . . . . . . 305
STRIG ON . . . . . . . . . . . . . . . . . . . . . . . . . 305
STRIG Trapping . . . . . . . . . . . . . . . . . 306-307
STRIG( ) OFF. . . . . . . . . . . . . . . . . . . . . . . 307
STRIG( ) ON . . . . . . . . . . . . . . . . . . . .228, 306
STRIG( ) STOP . . . . . . . . . . . . . . . . . .228, 306
String . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
compare. . . . . . . . . . . . . . . . . . . . . . . . . . . 56
concatenation. . . . . . . . . . . . . . . . . . . . . . . 55
constants . . . . . . . . . . . . . . . . . . . . . . . .38, 46
converting to . . . . . . . . . . . . . . . . . . . . . . 304
converting to ASCII . . . . . . . . . . . . . . . . . . 95
converting to numeric . . . . . . . . . .49, 50, 73, 216, 320
defining (DEFSTR) . . . . . . . . . . . . . . . . . . . 46
descriptor. . . . . . . . . . . . . . . . . . . . . . 354-355
expression . . . . . . . . . . . . . . . . . . . . . . . . . 33
operator . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
relations . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
storage requirements. . . . . . . . . . . . . . . . . 34
Structured programming techniques. . 169, 224, 283
Subroutine . . . . . . . . . . . . . . .169, 224, 349-352
Subscript . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
Substring . . . . . . . . . . .182, 191, 214, 215, 284
Subtraction . . . . . . . . . . . . . . . . . . . . . . .52, 55
SWAP. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 312


T


TAB . . . . . . . . . . . . . . . . . . . . . . 210, 261, 314
Tables (see Arrays) . . . . . . . . . . . . . . . . . 40-43
Tangent of a number. . . . . . . . . . . . . . . . . . 315
Text mode . . . . . . . . . . . . . . . . . . . . . . . . . . 291
Ticks, clock . . . . . . . . . . . . . . . . . . . . . 296-297
TIME$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 316
Time, setting, retrieving . . . . . . . . . . . . . . . 316
Trace function . . . . . . . . . . . . . . . . . . . . . . . 318
Trapping
mouse buttons. . . . . . . . . . . . . . . . . . . . . 228
Trigonometric Functions . . . . . . . . . . . . 342-343
TRON, TROFF . . . . . . . . . . . . . . . . . . . . . . . 318
Type Declaration Tags. . . . . . . . 40, 46, 72, 137


U


User Defined Functions. . . . . . . . . . . . . . . . 107
USR subroutine number . . . . . . . . . . . . . . . 141
USR . . . . . . . . . . . . . . . . . . . . . . . 319, 352-354
USR, argument of . . . . . . . . . . . . . . . . . . . . 321


V


VAL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 320
Variable
Accessing Strings . . . . . . . . . . . . . . . 354-355
address of . . . . . . . . . . . . . . . . . . . . . . . . 321
assigning variables . . . . . . . . . . . . . .175, 193
clearing . . . . . . . . . . . . . . . . . . . . . . . . . . 115
defining . . . . . . . . . . . . . . . . . . . . . . . . . . 137
exchanging values of. . . . . . . . . . . . . . . . 312
how BASIC classifies. . . . . . . . . . . . . . . . . 46
How BASIC stores. . . . . . . . . . . . . . . . . . 358
initilizing. . . . . . . . . . . . . . . . . . . . . . . . . 115
numeric . . . . . . . . . . . . . . . . . . . . . . . . .38, 47
rules for naming . . . . . . . . . . . . . . . . . . . . 39

string ........................... 38, 46
types ........................... 46
VARPTR ........................... 321, 356
VARPTR$ ........................... 322
Video Display Worksheet ........................... 344
Visual page ........................... 291


W


WAIT ........................... 323
WEND ........................... 324
WHILE ........................... 324
Whole numbers ........................... 159
WIDTH ........................... 117, 325
WRITE ........................... 328
WRITE# ........................... 329
Writing data
on the screen ........................... 328
to a sequential file ........................... 329


X


XOR ........................... 59, 270-271

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

| AUSTRALIA | BELGIUM | U. K. |
|---|---|---|
| 91 KURRAJONG ROAD<br>MOUNT DRUITT, N.S.W. 2770 | PARC INDUSTRIEL DE NANINNE<br>5140 NANINNE | BILSTON ROAD WEDNESBURY<br>WEST MIDLANDS WS10 7JN |

874-9485-08/84-SP

Printed in U.S.A.