



*Personal Computer  
Hardware Reference  
Library*

---

# BASIC

**Second Edition (May 1982)**

**Version 1.10**

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1981

# Preface

The IBM Personal Computer BASIC interpreter consists of three upward compatible versions: Cassette, Disk, and Advanced. This manual is a reference for all three versions of BASIC release 1.10. We shall use the general term "BASIC" in this book to refer to any of the versions of BASIC — Cassette, Disk, or Advanced.

The IBM Personal Computer BASIC Compiler is an optional software package available from IBM. If you have the BASIC Compiler, the *IBM Personal Computer Basic Compiler* manual is used in conjunction with this book for reference.

## How to Use This Manual

In order to use this manual, you should have some knowledge of general programming concepts; we are not trying to teach you how to program in this manual.

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

Chapter 1 is a brief overview of the three versions of IBM Personal Computer BASIC.

Chapter 2 tells you what you need to know to start using BASIC on your IBM Personal Computer. It tells you how to operate your computer using BASIC.

Chapter 3 covers a variety of topics which you will need to know before you actually start programming. Much of the information pertains to data representation when using BASIC. We discuss filenames here, along with many of the special input and output features available in IBM Personal Computer BASIC.

Chapter 4 is the reference section. It contains the syntax and semantics of every command, statement, and function in BASIC, ordered alphabetically.

The appendices contain other useful information, such as lists of error messages, ASCII codes, and math functions; and helpful information on machine language subroutines, diskette input and output, and communications. You can also find detailed information on more advanced subjects for the more experienced programmer.

We suggest you read through all of chapters 2 and 3 to become familiar with BASIC. Then you can refer to chapter 4 while you are actually programming to get information you need about each command or statement that you use.

## Syntax Diagrams

Each of the commands, statements, and functions described in this book has its syntax described according to the following conventions:

- Words in capital letters are keywords and must be entered as shown. They may be entered in any combination of uppercase and lowercase. BASIC always converts words to uppercase (unless they are part of a quoted string, remark, or DATA statement).
- You must supply any items in lowercase italic letters.
- Items in square brackets ( [ ] ) are optional.
- An ellipsis ( . . . ) indicates an item may be repeated as many times as you wish.
- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Let's look at an example:

`INPUT[;][“prompt”;] variable[,variable]...`

This says that for an INPUT statement to be valid, you must first have the keyword INPUT, followed optionally by a semicolon. Then, if you wish, you may include a *prompt* within quotation marks. If you do include the *prompt*, it must be followed by a semicolon. At least one *variable* is required for an INPUT statement. You may have more than one *variable* if you separate them with commas.

More detailed information on each of the parameters is included with the text accompanying the diagram. The information for this example is in Chapter 4, under “INPUT Statement.”

## Related Publications

The following manuals contain related information that you may find useful:

- The *IBM Personal Computer Guide to Operations* manual.
- The *IBM Personal Computer Disk Operating System* manual.
- The *IBM Personal Computer Technical Reference* manual.

# Summary of Changes

The following changes have been made in BASIC release 1.10:

- Any list to the screen or printer can be terminated by pressing Ctrl-Break.
- Printers (LPT1:, LPT2:, and LPT3:) may be opened in random mode. In release 1.00, these devices were always opened for sequential output, which would cause BASIC to add a line feed character after each carriage return character.

Opening a printer in random mode with a width of 255 suppresses the line feed after the carriage return, so that all characters may be sent to the printer without change. This mode can be used to support various types of graphics printers.

- the OPEN “COM... statement has the following new options:
  - RS        suppresses RTS (Request To Send)
  - CS[n]     controls CTS (Clear To Send)
  - DS[n]     controls DSR (Data Set Ready)
  - CD[n]     controls CD (Carrier Detect)
  - LF        sends a line feed following each carriage return

Also, a LEN=*number* option has been added to the OPEN “COM... statement to specify the maximum number of bytes which can be read from the file buffer when using GET or PUT. This option is included for compatibility with the BASIC Compiler.

- The STRIG function in Advanced BASIC now reads four joystick buttons. This is useful if you have four one-dimensional paddles.
- The VARPTR\$ function has been added. This keeps compatibility with the compiler, which needs this function for DRAW and PLAY.

# CONTENTS

<b>CHAPTER 1. THE VERSIONS OF BASIC . . . . .</b>	<b>1-1</b>
The Versions of BASIC .....	1-3
Cassette BASIC .....	1-4
Disk BASIC .....	1-5
Advanced BASIC .....	1-6
<b>CHAPTER 2. HOW TO START AND USE BASIC . . . . .</b>	<b>2-1</b>
Getting BASIC Started .....	2-3
Options on the BASIC Command .....	2-4
Modes of Operation .....	2-7
The Keyboard .....	2-8
Function Keys .....	2-9
Typewriter Keyboard .....	2-10
Numeric Keypad .....	2-16
Special Key Combinations .....	2-18
The BASIC Program Editor .....	2-20
Special Program Editor Keys .....	2-20
How to Make Corrections on the Current Line .....	2-33
Entering or Changing a BASIC Program .....	2-37
Changing Lines Anywhere on the Screen .....	2-39
Syntax Errors .....	2-41
<b>CHAPTER 3. GENERAL INFORMATION ABOUT PROGRAMMING IN BASIC . . . . .</b>	<b>3-1</b>
Line Format .....	3-3
Character Set .....	3-4
Reserved Words .....	3-6
Constants .....	3-9
Numeric Precision .....	3-11
Variables .....	3-12
How to Name a Variable .....	3-12
How to Declare Variable Types .....	3-13
Arrays .....	3-15
How BASIC Converts Numbers from One Precision to Another .....	3-18

<b>Numeric Expressions and Operators .....</b>	<b>3-21</b>
Arithmetic Operators .....	3-21
Relational Operators .....	3-23
Logical Operators .....	3-25
Numeric Functions .....	3-29
Order of Execution .....	3-29
<b>String Expressions and Operators .....</b>	<b>3-31</b>
Concatenation .....	3-31
String Functions .....	3-32
<b>Input and Output .....</b>	<b>3-33</b>
Files .....	3-33
Using the Screen .....	3-38
Other I/O Features .....	3-44

<b>CHAPTER 4. BASIC COMMANDS, STATEMENTS, FUNCTIONS, AND VARIABLES .....</b>	<b>4-1</b>
<b>How to Use This Chapter .....</b>	<b>4-3</b>
<b>Commands .....</b>	<b>4-6</b>
<b>Statements .....</b>	<b>4-8</b>
Non-I/O Statements .....	4-8
I/O Statements .....	4-13
<b>Functions and Variables .....</b>	<b>4-17</b>
Numeric Functions .....	4-17
String Functions .....	4-21
<b>ABS Function .....</b>	<b>4-23</b>
<b>ASC Function .....</b>	<b>4-24</b>
<b>ATN Function .....</b>	<b>4-25</b>
<b>AUTO Command .....</b>	<b>4-26</b>
<b>BEEP Statement .....</b>	<b>4-28</b>
<b>BLOAD Command .....</b>	<b>4-29</b>
<b>BSAVE Command .....</b>	<b>4-32</b>
<b>CALL Statement .....</b>	<b>4-34</b>
<b>CDBL Function .....</b>	<b>4-35</b>
<b>CHAIN Statement .....</b>	<b>4-36</b>
<b>CHR\$ Function .....</b>	<b>4-38</b>
<b>CINT Function .....</b>	<b>4-40</b>
<b>CIRCLE Statement .....</b>	<b>4-41</b>
<b>CLEAR Command .....</b>	<b>4-44</b>
<b>CLOSE Statement .....</b>	<b>4-46</b>
<b>CLS Statement .....</b>	<b>4-48</b>
<b>COLOR Statement .....</b>	<b>4-49</b>
The COLOR Statement in Text Mode ..	4-49
The COLOR Statement in Graphics Mode .....	4-54
<b>COM(n) Statement .....</b>	<b>4-56</b>

COMMON Statement .....	4-57
CONT Command .....	4-58
COS Function .....	4-60
CSNG Function .....	4-61
CSRLIN Variable .....	4-62
CVI, CVS, CVD Functions .....	4-63
DATA Statement .....	4-64
DATE\$ Variable and Statement .....	4-66
DEF FN Statement .....	4-68
DEF SEG Statement .....	4-71
DEFtype Statements .....	4-73
DEF USR Statement .....	4-75
DELETE Command .....	4-76
DIM Statement .....	4-77
DRAW Statement .....	4-79
EDIT Command .....	4-84
END Statement .....	4-85
EOF Function .....	4-86
ERASE Statement .....	4-87
ERR and ERL Variables .....	4-89
ERROR Statement .....	4-91
EXP Function .....	4-93
FIELD Statement .....	4-94
FILES Command .....	4-97
FIX Function .....	4-99
FOR and NEXT Statements .....	4-100
FRE Function .....	4-104
GET Statement (Files) .....	4-106
GET Statement (Graphics) .....	4-108
GOSUB and RETURN Statements .....	4-111
GOTO Statement .....	4-113
HEX\$ Function .....	4-115
IF Statement .....	4-116
INKEY\$ Variable .....	4-119
INP Function .....	4-121
INPUT Statement .....	4-122
INPUT # Statement .....	4-125
INPUT\$ Function .....	4-127
INSTR Function .....	4-129
INT Function .....	4-130
KEY Statement .....	4-131
KEY(n) Statement .....	4-134
KILL Command .....	4-136
LEFT\$ Function .....	4-137
LEN Function .....	4-138

LET Statement .....	4-139
LINE Statement .....	4-141
LINE INPUT Statement .....	4-144
LINE INPUT # Statement .....	4-145
LIST Command .....	4-147
LLIST Command .....	4-149
LOAD Command .....	4-150
LOC Function .....	4-153
LOCATE Statement .....	4-155
LOF Function .....	4-158
LOG Function .....	4-159
LPOS Function .....	4-160
<b>LPRINT and LPRINT USING</b>	
Statements .....	4-161
LSET and RSET Statements .....	4-163
MERGE Command .....	4-165
MID\$ Function and Statement .....	4-167
MKI\$, MKS\$, MKD\$ Functions .....	4-170
MOTOR Statement .....	4-172
NAME Command .....	4-173
NEW Command .....	4-174
OCT\$ Function .....	4-175
ON COM(n) Statement .....	4-176
ON ERROR Statement .....	4-178
<b>ON...GOSUB and ON...GOTO</b>	
Statements .....	4-180
ON KEY(n) Statement .....	4-182
ON PEN Statement .....	4-185
ON STRIG(n) Statement .....	4-187
OPEN Statement .....	4-189
OPEN "COM..." Statement .....	4-194
OPTION BASE Statement .....	4-200
OUT Statement .....	4-201
PAINT Statement .....	4-203
PEEK Function .....	4-205
PEN Statement and Function .....	4-206
PLAY Statement .....	4-209
POINT Function .....	4-213
POKE Statement .....	4-214
POS Function .....	4-215
PRINT Statement .....	4-216
PRINT USING Statement .....	4-219
<b>PRINT # and PRINT # USING</b>	
Statements .....	4-225

PSET and PRESET Statements .....	4-228
PUT Statement (Files) .....	4-230
PUT Statement (Graphics) .....	4-232
RANDOMIZE Statement .....	4-236
READ Statement .....	4-238
REM Statement .....	4-240
RENUM Command .....	4-241
RESET Command .....	4-243
RESTORE Statement .....	4-244
RESUME Statement .....	4-245
RETURN Statement .....	4-247
RIGHT\$ Function .....	4-248
RND Function .....	4-249
RUN Command .....	4-251
SAVE Command .....	4-253
SCREEN Function .....	4-255
SCREEN Statement .....	4-257
SGN Function .....	4-260
SIN Function .....	4-261
SOUND Statement .....	4-262
SPACE\$ Function .....	4-265
SPC Function .....	4-266
SQR Function .....	4-267
STICK Function .....	4-268
STOP Statement .....	4-270
STR\$ Function .....	4-272
STRIG Statement and Function .....	4-273
STRIG(n) Statement .....	4-275
STRING\$ Function .....	4-276
SWAP Statement .....	4-277
SYSTEM Command .....	4-278
TAB Function .....	4-279
TAN Function .....	4-280
TIME\$ Variable and Statement .....	4-281
TRON and TROFF Commands .....	4-283
USR Function .....	4-284
VAL Function .....	4-285
VARPTR Function .....	4-286
VARPTR\$ Function .....	4-288
WAIT Statement .....	4-290
WHILE and WEND Statements .....	4-292
WIDTH Statement .....	4-294
WRITE Statement .....	4-298
WRITE # Statement .....	4-299

<b>APPENDIX A. MESSAGES .....</b>	<b>A-5</b>
<b>APPENDIX B. BASIC DISKETTE INPUT AND OUTPUT .....</b>	<b>B-1</b>
Specifying Filenames .....	B-2
Commands for Program Files .....	B-2
Diskette Data Files - Sequential and Random I/O .....	B-4
Sequential Files .....	B-4
Random Files .....	B-8
Performance Hints .....	B-15
<b>APPENDIX C. MACHINE LANGUAGE SUBROUTINES .....</b>	<b>C-1</b>
Setting Memory Aside for Your Subroutines .....	C-2
Getting the Subroutine Code into Memory .....	C-3
Poking a Subroutine into Memory ....	C-4
Loading the Subroutine from a File ...	C-5
Calling the Subroutine from Your BASIC Program .....	C-8
Common Features of CALL and USR ...	C-8
CALL Statement .....	C-10
USR Function Calls .....	C-14
<b>APPENDIX D. CONVERTING PROGRAMS TO IBM PERSONAL COMPUTER BASIC .....</b>	<b>D-1</b>
File I/O .....	D-1
Graphics .....	D-1
IF...THEN .....	D-2
Line Feeds .....	D-3
Logical Operations .....	D-3
MAT Functions .....	D-4
Multiple Assignments .....	D-4
Multiple Statements .....	D-4
PEEKS and POKEs .....	D-4
Relational Expressions .....	D-5
Remarks .....	D-5
Rounding of Numbers .....	D-5
Sounding the Bell .....	D-5
String Handling .....	D-6
Use of Blanks .....	D-7
Other .....	D-7

<b>APPENDIX E. MATHEMATICAL FUNCTIONS .....</b>	<b>E-1</b>
<b>APPENDIX F. COMMUNICATIONS .....</b>	<b>F-1</b>
Opening a Communications File .....	F-1
Communication I/O .....	F-1
An Example Program .....	F-4
<b>Operation of Control Signals .....</b>	<b>F-6</b>
Control of Output Signals with OPEN ...	F-6
Use of Input Control Signals .....	F-7
Testing for Modem Control Signals ...	F-7
Direct Control of Output Control Signals .....	F-8
Communication Errors .....	F-10
<b>APPENDIX G. ASCII CHARACTER CODES .....</b>	<b>G-1</b>
Extended Codes .....	G-6
<b>APPENDIX H. HEXADECIMAL CONVERSION TABLE .....</b>	<b>H-1</b>
<b>APPENDIX I. TECHNICAL INFORMATION AND TIPS .....</b>	<b>I-1</b>
Memory Map .....	I-2
How Variables Are Stored .....	I-3
BASIC File Control Block .....	I-4
Keyboard Buffer .....	I-7
Search Order for Adapters .....	I-7
Switching Displays .....	I-8
Some Techniques with Color .....	I-9
Tips and Techniques .....	I-10
<b>APPENDIX J. GLOSSARY .....</b>	<b>J-1</b>
<b>INDEX .....</b>	<b>X-1</b>

# NOTES

# CHAPTER 1. THE VERSIONS OF BASIC

## Contents

The Versions of BASIC .....	1-3
Cassette BASIC .....	1-4
Disk BASIC .....	1-5
Advanced BASIC .....	1-6

# NOTES

# The Versions of BASIC

The IBM Personal Computer offers three different versions of the BASIC interpreter:

- Cassette
- Disk
- Advanced

The three versions of BASIC are upward compatible; that is, Disk BASIC does everything Cassette BASIC does, plus a little more, and Advanced BASIC does everything Disk BASIC does, plus a little more. The differences between the versions are discussed in more detail below.

The BASIC commands, statements, and functions for all three versions of the BASIC interpreter are described in detail in "Chapter 4. BASIC Commands, Statements, Functions, and Variables." Included in each description is a section called **Versions:**, where we tell you which versions of BASIC support the command, statement, or function.

For example, if you look under "CHAIN Statement" in Chapter 4, you will note that it says:

<b>Versions:</b>	Cassette	Disk	Advanced	Compiler
		***	***	(**)

The asterisks indicate which versions of BASIC support the statement. This example shows that you can use the CHAIN statement for programs written in the Disk and Advanced versions of BASIC.

In this example you will notice that the asterisks under the word "Compiler" are in parentheses. This means that there are differences between the way the statement works under the BASIC interpreter and the way it works under the IBM Personal Computer BASIC Compiler. The IBM Personal Computer BASIC Compiler is an optional software package available from IBM. If you have the BASIC Compiler, the *IBM Personal Computer BASIC Compiler* manual explains these differences.

# Cassette BASIC

The nucleus of BASIC is the Cassette version, which is built into your IBM Personal Computer in 32K-bytes of read-only storage. You can use Cassette BASIC on an IBM Personal Computer with any amount of random access memory. The amount of storage you can use for such things as programs and data depends on how much memory you have in your IBM Personal Computer. The number of "bytes free" will be displayed after you switch on the computer.

The only storage device you can use to save information in Cassette BASIC is a cassette tape recorder. You cannot use diskettes with Cassette BASIC.

Some special features you will find in this and the other two versions of BASIC are:

- An extended character set of 256 different characters which can be displayed. In addition to the usual letters, numbers, and special symbols, you also have international characters like ñ, â, and ï. You will also find symbols which are commonly used in scientific and mathematical applications, such as Greek letters. There are also a variety of other symbols.
- Graphics capability. If you have the Color/Graphics Monitor Adapter, you can draw points, lines, and even entire pictures. The screen can be *all points addressable* in either medium or high resolution. More information on this can be found in Chapter 3.
- Special input/output devices. The IBM Personal Computer has a speaker which you can use to make sound. Also, BASIC supports a light pen and joysticks which help make your programs more interesting as well as more fun.

# Disk BASIC

This version of BASIC comes as a program on the IBM Personal Computer Disk Operating System (DOS) diskette. DOS is a separate product available from IBM. You have to load Disk BASIC into memory before you can use it. Disk BASIC requires a diskette-based machine with at least 32K-bytes of random access memory. The amount of storage you can use for such things as programs and data is displayed on the screen when you start BASIC.

Special features of Disk BASIC are:

- Input/output to diskette in addition to cassette. See "Appendix B. BASIC Diskette Input and Output" for special considerations when using diskette files.
- An internal "clock," which keeps track of the date and time.
- Asynchronous communications (RS232) support, which you can use if you have an Asynchronous Communications Adapter. Refer to "Appendix F. Communications" for details.
- Support for two additional printers.

# Advanced BASIC

Advanced BASIC, the most extensive form of BASIC available on the IBM Personal Computer, does everything that Cassette and Disk BASIC do, and more. Like Disk BASIC, it is a program on the IBM DOS diskette which you must load into memory to use. Advanced BASIC requires a diskette-based machine with at least 48K-bytes of random access memory. As with the other versions, the number of free bytes you will have for programs and data is displayed on the screen when you start BASIC.

Key features found only in Advanced BASIC are the following:

- Event trapping. A program can respond to the occurrence of a specific event by "trapping" (automatically branching) to a specific program line. Events include: communications activity, a function key being pressed, the button being pressed on a joystick, and the light pen being activated.
- Advanced graphics. Additional statements are CIRCLE, PUT, GET, PAINT, and DRAW. These operations make it easier to create more complex graphics with the Color/Graphics Monitor Adapter.
- Advanced music support. The PLAY statement allows easy usage of the built-in speaker to create musical tones.

# CHAPTER 2. HOW TO START AND USE BASIC

## Contents

Getting BASIC Started .....	2-3
Options on the BASIC Command .....	2-4
Modes of Operation .....	2-7
The Keyboard .....	2-8
Function Keys.....	2-9
Typewriter Keyboard.....	2-10
Special Symbols .....	2-11
Uppercase .....	2-12
Backspace .....	2-12
PrtSc .....	2-13
Other Shifts .....	2-13
Numeric Keypad .....	2-15
Keypad Shift .....	2-16
Special Key Combinations .....	2-17
The BASIC Program Editor .....	2-19
Special Program Editor Keys .....	2-19
How to Make Corrections on the	
Current Line .....	2-32
Changing Characters .....	2-32
Erasing Characters .....	2-33
Adding Characters .....	2-34
Erasing Part of a Line .....	2-35
Cancelling a Line .....	2-35
Entering or Changing a BASIC	
Program .....	2-36
Adding a New Line to the	
Program .....	2-36
Replacing or Changing an Existing	
Program Line .....	2-37
Deleting Program Lines .....	2-37
Deleting an Entire Program .....	2-38
Changing Lines Anywhere on the	
Screen .....	2-38
Syntax Errors .....	2-40

# **NOTES**

# Getting BASIC Started

It's easy to start BASIC on the IBM Personal Computer:

## To Start Cassette BASIC:

Just switch the computer on. If your system has diskette drives, you should make sure you don't have a diskette in drive A, or leave the drive door open.

The words "Version C" and the release number will be displayed along with the number of free bytes you have available.

## To Start Disk BASIC:

1. Start DOS. To do this, you can:
  - a. Insert the IBM DOS diskette in drive A:.
  - b. Switch on the computer.
2. Enter the command **BASIC** when DOS prompts you for a command.

The words "Version D" and the release number will be displayed along with the number of free bytes.

## To Start Advanced BASIC:

1. Start DOS as described above.
2. Enter the command **BASICA** in response to the DOS prompt.

The words "Version A" and the release number will be displayed along with the number of free bytes.

# Options on the BASIC Command

You can include options on the BASIC or BASICA command when you start Disk or Advanced BASIC. These options specify the amount of storage BASIC uses to hold programs and data, and for buffer areas. You can also ask BASIC to immediately load and run a program.

These options are not required—BASIC will work just fine without them. So if you're new to BASIC, you may wish to skip over this section and go on to the next section, "Modes of Operation." Then you can refer back to this section when you become more familiar with BASIC and its capabilities.

The complete format of the BASIC command is:

```
BASIC[A] [filespec] [/F:files] [/S:bsize]  
[/C:combuffer] [/M:max workspace]
```

*filespec* is the file specification of a program to be loaded and executed immediately. It must be a character string constant, but it should *not* be enclosed in quotation marks. It should conform to the rules for specifying files described under "Naming Files" in "Chapter 3. General Information about Programming in BASIC." A default extension of .BAS is used if none is supplied and the length of the filename is eight characters or less. If you include *filespec*, BASIC proceeds as if a RUN*filespec* command were the first thing you entered once BASIC is ready. Note that when you specify *filespec*, the BASIC heading with the copyright notices is not displayed.

/F:*files* sets the maximum number of files that may be open at any one time during the execution of a BASIC program. Each file requires 188 bytes of memory for the file control block, plus the buffer size specified in the /S: option. If the /F: option is omitted, the number of files defaults to three. The maximum value is 15.

*/S:bsize* sets the buffer size for use with random files. The record length parameter on the OPEN statement may not exceed this value. The default buffer size is 128 bytes. The maximum value you may enter is 32767. We suggest you use */S:512* for improved performance when using random files.

*/C:combuffer* sets the size of the buffer for receiving data when using the Asynchronous Communications Adapter. This option has no effect unless you have an Asynchronous Communications Adapter on your system. The buffer for transmitting data with communications is always allocated to 128 bytes. The maximum value you may enter for the */C:* option is 32767. If the */C:* option is omitted, 256 bytes are allocated for the receive buffer. If you have a high-speed line, we suggest you use */C:1024*. If you have two Asynchronous Communications Adapters on your system, both receive buffers are set to the size specified by this option. You may disable RS232 support by using a value of zero (*/C:0*), in which case no buffer space will be reserved for communications, and communications support will not be included when BASIC is loaded.

*/M:max workspace* sets the maximum number of bytes that may be used as BASIC workspace. BASIC is only able to use a maximum of 64K-bytes of memory, so the highest value you may set is 64K (hex FFFF). You can use this option in order to reserve space for machine language subroutines or for special data storage. You may wish to refer to "Memory Map" in Appendix I for more detailed information on how BASIC uses memory. If the */M:* option is omitted, all available memory up to a maximum of 64K-bytes is used.

**Note:** *files*, *max workspace*, *bsize*, and *combuffer* are all numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Some examples of using the BASIC command:

BASIC PAYROLL.BAS

This will start Disk BASIC so that it will use the defaults as just described — all memory and three files. The program PAYROLL.BAS will be loaded and executed.

BASIC A:INVEN/F:6

Here we start Advanced BASIC to use all memory and six files, and load and execute INVEN.BAS. Remember, .BAS is the default extension.

BASIC /M:32768

This command starts Disk BASIC so the maximum workspace size is 32768. That is, BASIC will use only 32K-bytes of memory. No more than three files will be used at one time.

BASIC B:CHKWRR.TST/F:2/M:&H9000

This command sets the maximum workspace size to hex 9000. This means Advanced BASIC will be able to use up to 36K-bytes of memory. Also, file control blocks are set up for two files, and the program CHKWRR.TST on the diskette in drive B is loaded and executed.

# Modes of Operation

Once BASIC is started, it displays the prompt **Ok**. **Ok** means BASIC is ready for you to tell it what to do. Sometimes this state is known as *command level*. At this point, you may talk to BASIC in either of two modes: the *direct mode* or the *indirect mode*.

## Indirect Mode

You enter programs using indirect mode. To tell BASIC the line you are entering is part of a program, you begin the line with a *line number*. The line is then stored as part of the program in memory. The program in storage can be executed by entering the **RUN** command. For example:

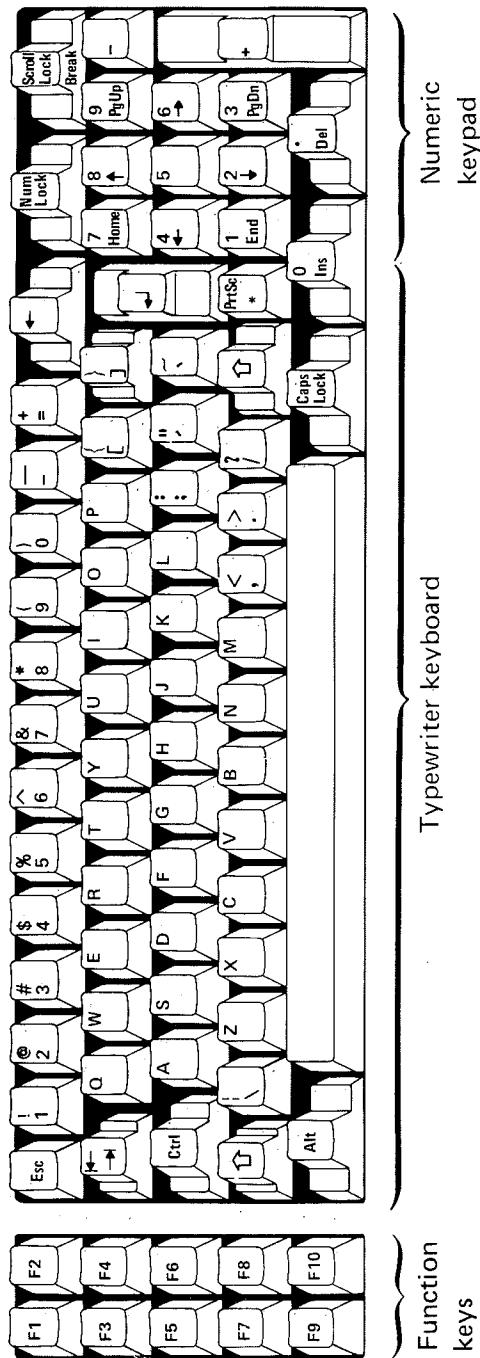
```
Ok  
1 PRINT 20+2  
RUN  
22  
Ok
```

## Direct Mode

Direct mode means you are telling BASIC to perform your request immediately after the request is entered. You tell BASIC to do this by *not* preceding the statement or command with a line number. You can display results of arithmetic and logical operations immediately or store them for later use, but the instructions themselves are not saved after they are executed. This mode is useful for debugging as well as for quick computations that do not require a complete program. For example:

```
Ok  
PRINT 20+2  
22  
Ok
```

# The Keyboard

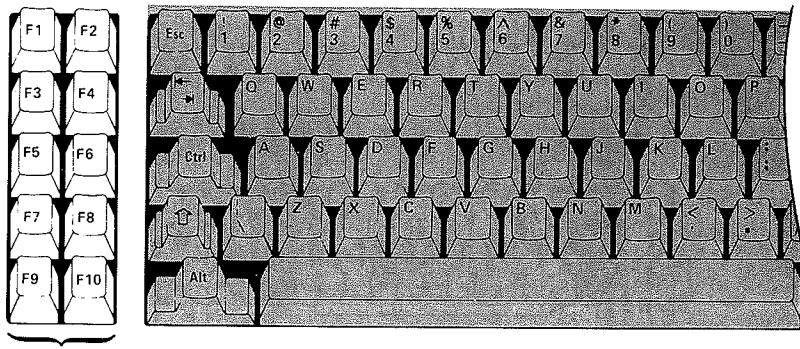


The keyboard is divided into three general areas:

- Ten function keys, labeled F1 through F10, are on the left side of the keyboard.
- The “typewriter” area is in the middle. This is where you find the regular letter and number keys.
- The numeric keypad, similar to a calculator keyboard, is on the right side.

All the keys, in all three areas of the keyboard, are typematic. That means they repeat as long as you hold them down. Each of the keyboard areas are explained in more detail below:

## Function Keys

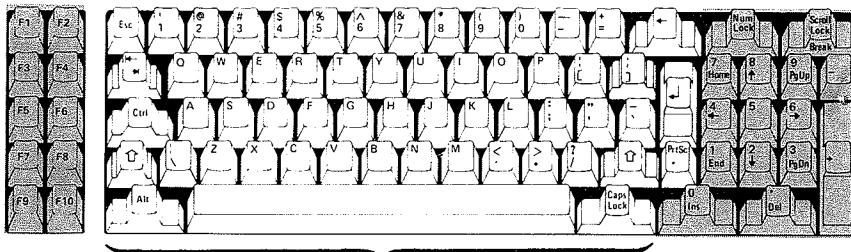


Function  
Keys

The function keys can be used:

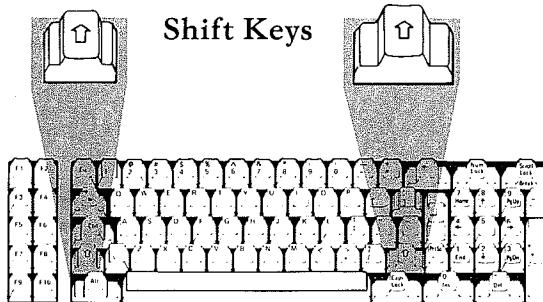
- As “soft keys.” That is, you can set each key to automatically type any sequence of characters. In fact, some frequently-used commands have already been assigned to these keys. You may change these if you wish. Refer to “KEY Statement” in Chapter 4 for details.
- As program interrupts in Advanced BASIC, through use of the ON KEY statement. See “ON KEY(n) Statement” in Chapter 4.

# Typewriter Keyboard

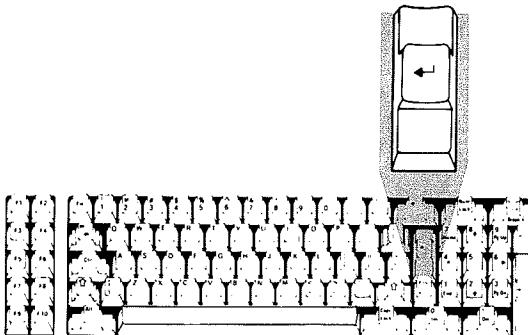


Typewriter Keyboard

The typewriter area of the keyboard behaves much like a standard typewriter. All the letters are there, in their usual places. The numbers 0 through 9 are on the top row, along with some special characters.



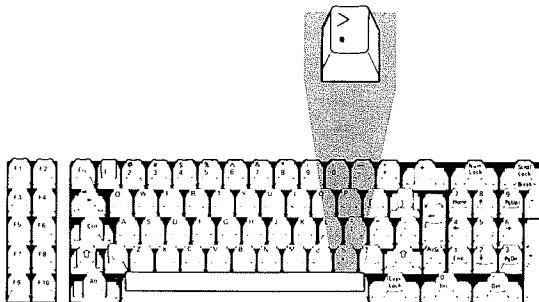
Capital letters and the special characters shown above the numbers on the number keys are displayed by holding down either of the Shift keys and pressing the desired key.



The key with the ↵ symbol on it is the carriage return key. You usually have to press this key to enter information into the computer. We will refer to it as the *Enter* key from now on.

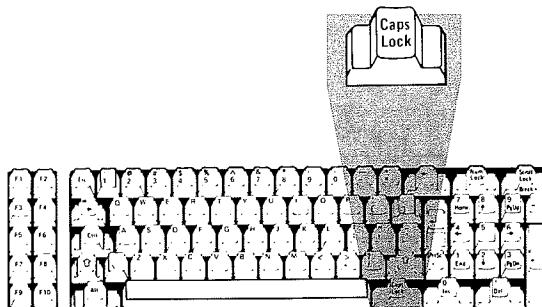
There are several important differences between this keyboard and a regular typewriter, however.

### Special Symbols:



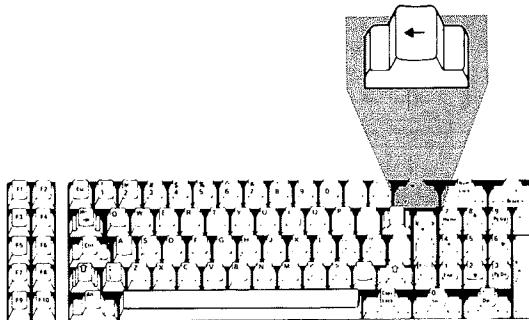
This keyboard has some special symbols that you won't find on a regular typewriter, like ^, [, and ]; and some characters are not where you might expect them to be if you're used to using a typewriter. For example, the uppershift period (.) is not a period, but the > symbol.

## Uppercase:

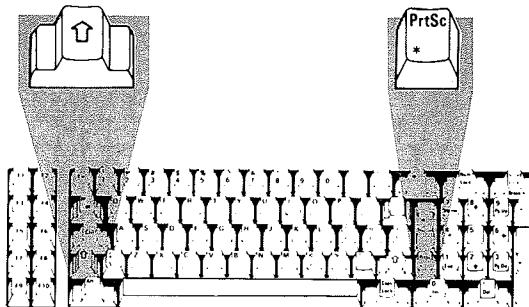


This keyboard does not have a normal Shift Lock key. The Caps Lock key is similar to a Shift Lock key, but it only gives you capital letters, and will not give you the uppershift characters on the numeric or other keys. After you press this key, you will continue to get capital letters until you press it again. You can get lowercase letters when in Caps Lock state by pressing and holding one of the Shift keys. When you release the Shift key, you'll go back to Caps Lock state.

## Backspace:

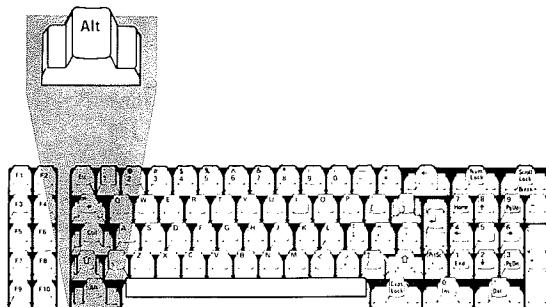


The Backspace key behaves somewhat differently from the Backspace key on a typewriter. It not only backspaces, it erases what you've typed as well. You should use the Cursor Left key to avoid erasing what you've typed. Refer to "The BASIC Program Editor" later in this chapter.



Below the Enter key is a key labeled PrtSc on top and \* on the bottom. "PrtSc" stands for "Print screen." When the keyboard is in lowershift, pressing this key causes an asterisk to be typed. In uppershift, however, this is a special key that causes a copy of what is on the screen to be printed on the printer (LPT1:). So, if you ever need a hard (printed) copy of what is currently being displayed, just press and hold one of the Shift keys, and press the PrtSc key. (Note: Characters which are unrecognizable by the printer are printed as blanks.)

**Other Shifts:** In addition to the Shift keys which change the keyboard from lowershift to uppershift, there are two other "shift" keys on the typewriter keyboard. They are the Alt (Alternate) and the Ctrl (Control) keys. You use both of these keys like the Shift keys; that is, you press and hold the Alt (or Ctrl) key, then press the desired key. Then you can release both keys. However, Alt and Ctrl cause different things to happen.

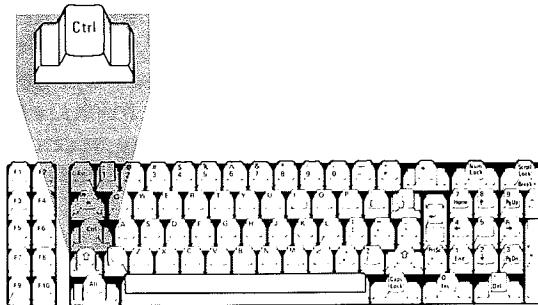


The Alt key enables easy entry of BASIC statement keywords. This key allows you to type an entire BASIC keyword with a single keystroke.

The BASIC keyword is typed when the Alt key is held down while one of the alphabetic keys A-Z is pressed. Keywords associated with each letter are summarized below. Letters not having reserved words are noted by "(no word)".

A	AUTO	N	NEXT
B	BSAVE	O	OPEN
C	COLOR	P	PRINT
D	DELETE	Q	(no word)
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX\$	U	USING
I	INPUT	V	VAL
J	(no word)	W	WIDTH
K	KEY	X	XOR
L	LOCATE	Y	(no word)
M	MOTOR	Z	(no word)

The Alt key is also used with the keys on the numeric keypad to enter characters not found on the keys. This is done by holding down the Alt key and typing the three-digit ASCII code for the character. (See "Appendix G. ASCII Character Codes" for a complete list of ASCII codes.)

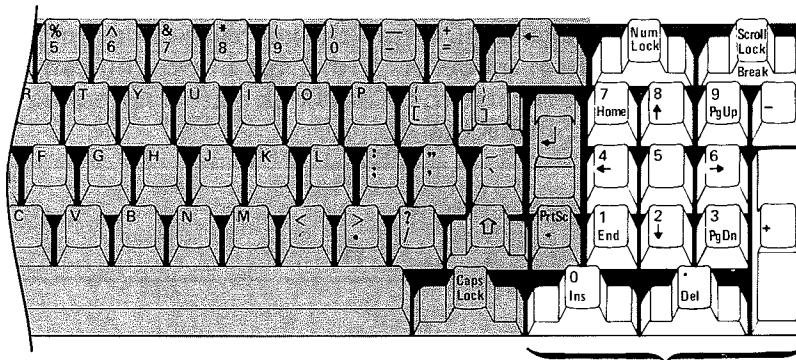


The Ctrl key is also used to enter certain codes and characters not otherwise available from the keyboard.

For example, **Ctrl-G** is the *bell* character. When this character is printed, the speaker beeps. Note how the notation “Ctrl-G” means you press and hold the Ctrl key, then press the G key. Then you can release both keys.

You also use the Ctrl key together with other keys when you edit programs with the program editor.

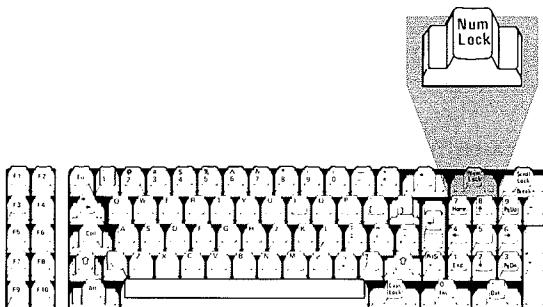
## Numeric Keypad



Numeric Keypad

Usually you will be using the numeric keypad keys for their functions with the program editor. These keys allow you to move the cursor up, down, right, and left. You can insert and delete characters using these keys. Refer to the following section, "The BASIC Program Editor," for complete information.

**Note:** The Scroll Lock, Pg Up, and Pg Dn keys are not used by BASIC, but they may be given meaning within a program.

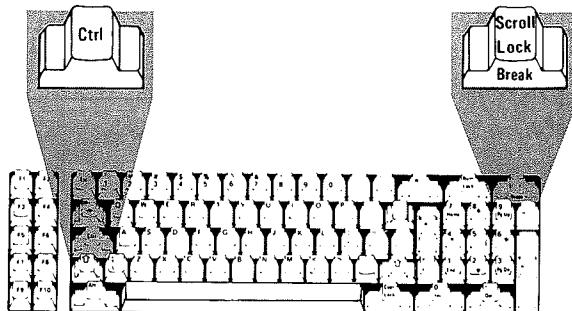


**Keypad Shift:** You can use the Num Lock key to set the numeric keypad so it works more like a calculator keypad. Pressing the Num Lock key shifts the numeric keypad into its own uppershift mode, so that you get the numbers 0 through 9 and the decimal point, as indicated on the keytops. Pressing Num Lock again will return the keypad to its normal cursor control mode. As with Caps Lock, you can temporarily reverse the Num Lock state by pressing one of the Shift Keys.

# Special Key Combinations

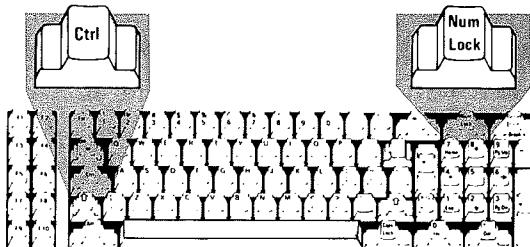
You should be aware of the special functions of the following combinations of keys:

## Ctrl-Break



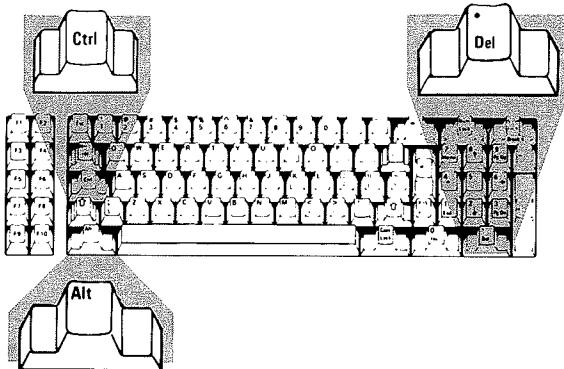
Ctrl-Break interrupts program execution at the next BASIC instruction and returns to BASIC command level. It is also used to exit AUTO line numbering mode.

## Ctrl-Num Lock



Ctrl-Num Lock sends the computer into a *pause* state. This can be used to temporarily halt printing or program listing. The pause continues until any key other than the "shift" keys, the Break key, and the Ins key, is pressed. (See "Uppercase," "Other Shifts," and "Keypad Shift" earlier in this section.)

## Alt-Ctrl-Del



If the computer power is on, Alt-Ctrl-Del performs a *System Reset*. In other words, it's similar to switching the computer from off to on. You must press the Ctrl and Alt keys (in either order) and hold them down, then press the Del key. Then you can release all three keys. Doing a System Reset with these keys is preferable to flipping the power switch off and on again, because the system will start faster.

# The BASIC Program Editor

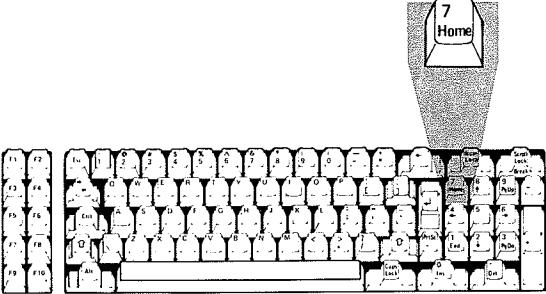
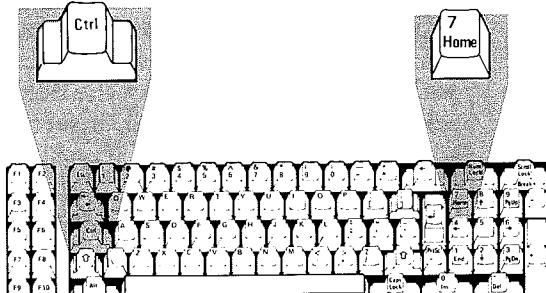
Any line of text typed while BASIC is at command level is processed by the BASIC program editor. The program editor is a "screen line editor." That is, you can change a line anywhere on the screen, but you can only change one line at a time. The change will only take effect if you press Enter on that line.

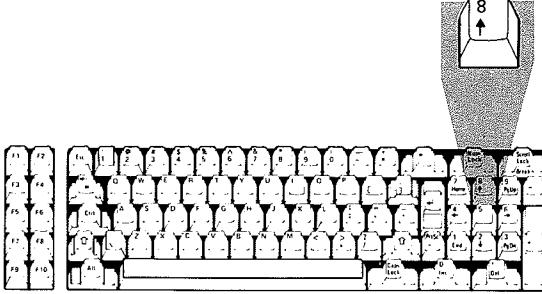
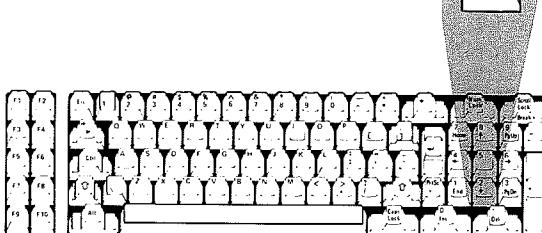
Use of the program editor can save a lot of time during program development. To become familiar with its features, we suggest you enter a sample program and practice all the editing capabilities. The best way for you to get a "feel" for the editing process is to try editing a few lines while studying the information that follows.

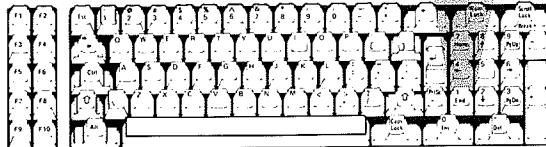
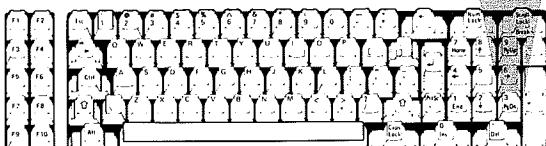
As you type things on your computer, you'll notice a blinking underline or box appearing just to the right of the last character you typed. This line or box is called the *cursor*. It marks the next position at which a character is to be typed, inserted, or deleted.

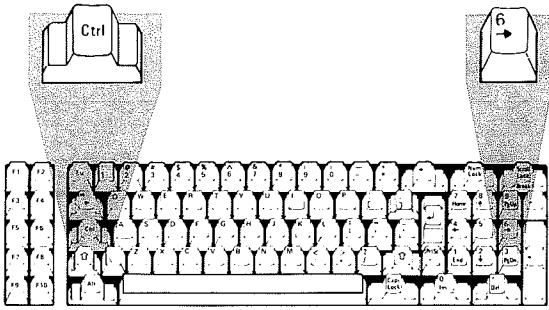
## Special Program Editor Keys

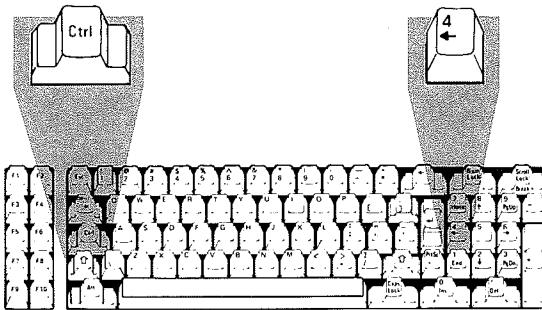
You use the keys on the numeric keypad, the Backspace key, and the Ctrl key to move the cursor to a location on the screen, insert characters, or delete characters. The keys and their functions are listed on the next pages.

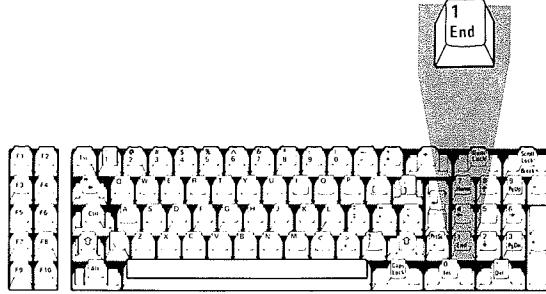
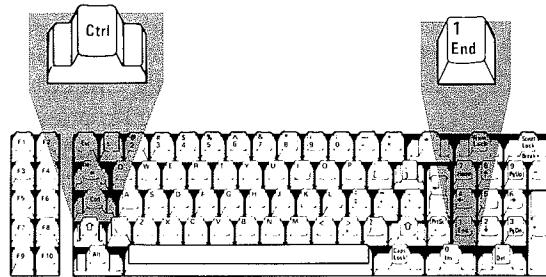
Key(s)	Function
<b>Home</b>	 <p>Moves the cursor to the upper left-hand corner of the screen.</p>
<b>Ctrl-Home</b>	 <p>Clears the screen and positions the cursor in the upper left-hand corner of the screen.</p>

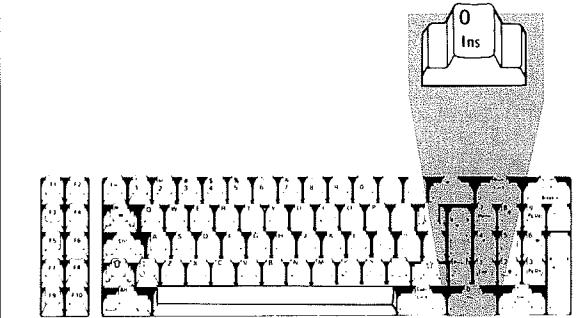
Key(s)	Function
↑ <b>(Cursor Up)</b>	 Moves the cursor one position up.
↓ <b>(Cursor Down)</b>	 Moves the cursor one position down.

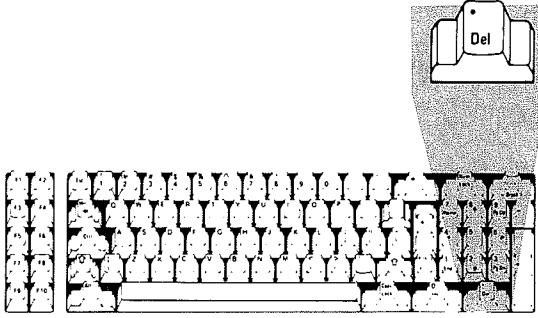
Key(s)	Function
<b>←</b> <b>(Cursor Left)</b>	  <p>Moves the cursor one position left. If the cursor advances beyond the left edge of the screen, the cursor will move to the right side of the screen on the preceding line.</p>
<b>→</b> <b>(Cursor Right)</b>	  <p>Moves the cursor one position right. If the cursor advances beyond the right edge of the screen, the cursor will move to the left side of the screen on the next line down.</p>

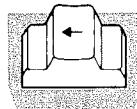
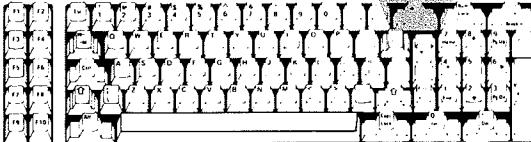
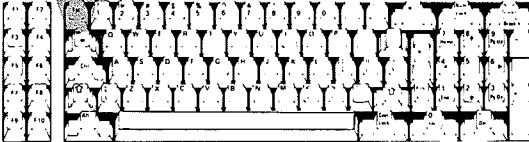
Key(s)	Function
<b>Ctrl- →</b> <b>(Next Word)</b>	 <p>Moves the cursor right to the next <i>word</i>. A word is defined as a character or group of characters which begins with a letter or number. Words are separated by blanks or special characters. So, the next word will be the next letter or number to the right of the cursor which follows a blank or special character.</p> <p>For example, suppose we have the following line:</p> <pre>LINE (L1,LOW2)-(MAX,48) ,3 , BF</pre> <p>As you can see, the cursor is presently in the middle of the word LOW2. If we press Next Word (Ctrl-Cursor Right), the cursor will move to the beginning of the next word, which is MAX:</p> <pre>LINE (L1,LOW2)-(MAX,<u>48</u>) ,3 , BF</pre> <p>If we press Next Word again, the cursor will move to the next word, which is the number 48:</p> <pre>LINE (L1,LOW2)-(MAX,<u>48</u>) ,3 , BF</pre>

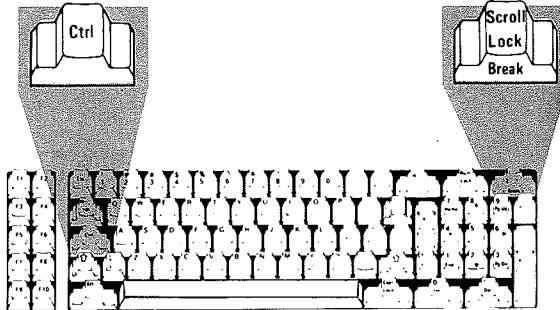
Key(s)	Function
<b>Ctrl-←</b> <b>(Previous Word)</b>	 <p>Moves the cursor left to the previous word. The previous word will be the letter or number to the left of the cursor which is preceded by a blank or special character.</p> <p>For example, suppose we have:</p> <pre>LINE (L1,LOW2)-(MAX,48) ,3 , BF_</pre> <p>If we press Previous Word (Ctrl-Cursor Left), the cursor moves to the beginning of the word BF:</p> <pre>LINE (L1,LOW2)-(MAX,48) ,3 , <u>BF</u></pre> <p>When we press Previous Word again, the cursor moves to the previous word, which is the number 3:</p> <pre>LINE (L1,LOW2)-(MAX,48) ,<u>3</u> , BF</pre> <p>And if we press it twice more, the cursor will back up first to the number 48, then to the word MAX:</p> <pre>LINE (L1,LOW2)-(MAX,48) ,<u>3</u> , BF</pre>

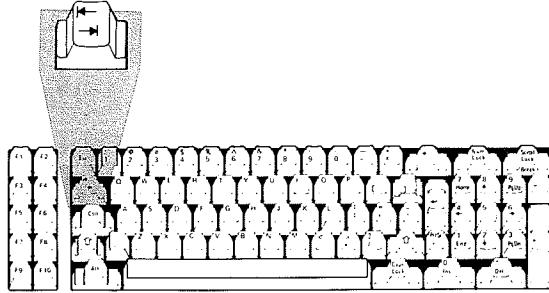
Key(s)	Function
<b>End</b>	 <p>Moves the cursor to the end of the logical line. Characters typed from this position are added to the end of the line.</p>
<b>Ctrl-End</b>	 <p>Erases to the end of logical line from the current cursor position. All physical screen lines are erased until the terminating Enter is found.</p>

Key(s)	Function
Ins	 <p>Sets insert mode. If insert mode is off, then pressing this key will turn it on. If insert mode is already on, then you will turn it off when you press this key. When you're in insert mode, the cursor covers the lower half of the character position.</p> <p>When insert mode is on, characters above and following the cursor move to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding occurs; that is, as characters advance off the right side of the screen they return on the left on a subsequent line.</p> <p>When insert mode is off, any characters typed replace existing characters on the line.</p> <p>Besides pressing the Ins key again, insert mode will also be turned off when you press any of the cursor movement keys or the Enter key.</p>

Key(s)	Function
Del	 <p>Deletes the character at the current cursor position. All characters to the right of the deleted character move one position left to fill in the empty space. Line folding occurs; that is, if a logical line extends beyond one physical line, characters on subsequent lines move left one position to fill in the previous space, and the character in the first column of each subsequent line moves up to the end of the preceding line.</p>

Key(s)	Function
 (Backspace)	  <p>Deletes the last character typed. That is, it deletes the character to the left of the cursor. All characters to the right of the deleted character move left one position to fill in the space. Subsequent characters and lines within the current logical line move up as with the Del key.</p>
Esc	  <p>When pressed anywhere in the line, erases the entire logical line from the screen. The line is not passed to BASIC for processing. If it is a program line, it is not erased from the program in memory.</p>

Key(s)	Function
Ctrl-Break	 <p>Returns to command level, <i>without</i> saving any changes that were made to the current line being edited. It does not erase the line from the screen like Esc does.</p>

Key(s)	Function
<p>→ (Tab)</p> 	<p>Moves the cursor to the next tab stop. Tab stops occur every eight character positions; that is, at positions 1, 9, 17, etc.</p> <p>When insert mode is off, pressing the Tab key moves the cursor over characters until it reaches the next tab stop.</p> <p>For example, suppose we have the following line:</p> <pre data-bbox="311 905 670 938">1Ø REM this is a remark</pre> <p>If we press the Tab key, the cursor will move to the ninth position as shown:</p> <pre data-bbox="311 1068 670 1101">1Ø REM this is a <u>remark</u></pre> <p>If we press the Tab key again, the cursor moves to the 17th position on the line:</p> <pre data-bbox="311 1232 670 1264">1Ø REM this is a <u>remark</u></pre>

Key(s)	Function
→ (Tab)	<p>(continued)</p> <p>When insert mode is on, pressing the Tab key inserts blanks from the current cursor position to the next tab stop. Line folding occurs as explained under <b>Ins</b>.</p> <p>For example, suppose we have this line:</p> <p>10 REM <u>this</u> is a remark</p> <p>If we press the Ins key and then the Tab key, blanks are inserted up to position 17:</p> <p>10 REM th       <u>is</u> a remark</p>

## How to Make Corrections on the Current Line

Since any line of text typed while BASIC is at command level is processed by the program editor, you can use any of the keys described in the previous section under "Special Program Editor Keys."

BASIC is always at command level after the prompt **Ok** and until a **RUN** command is given.

A *logical line* is a string of text which BASIC treats as a unit. It is possible to extend a logical line over more than one physical screen line by simply typing beyond the edge of the screen. The cursor will *wrap* to the next screen line. You can also use a line feed (**Ctrl-Enter**). Typing a line feed causes subsequent text to be printed on the next screen line without your having to enter all the blanks to move the cursor there. The line is not processed; this only happens when you press **Enter**.

Note that the line feed actually causes the remainder of the physical screen line to be filled with blank characters. A line feed character is not added to the text. These blanks are included in the 255 characters allowed for a BASIC line.

When the **Enter** key is finally pressed, the entire logical line is passed to BASIC for processing.

**Changing Characters:** If you are typing a line and discover you typed something incorrectly, you can correct it. Use the **Cursor Left** or other cursor movement keys to move the cursor to the position where the mistake occurred, and type the correct letters on top of the wrong ones. Then you can move the cursor back to the end of the line using the **Cursor Right** or **End** keys, and continue typing.

For example, suppose we have typed the following:

LOAD "V;PROG\_

We accidentally typed V; instead of B:. We fix the problem by pressing Previous Word (Ctrl-Cursor Left) twice, until the cursor is under the V:

LOAD "V;PROG

Then we type B::

LOAD "B:PROG

Then we press the End key:

LOAD "B:PROG\_

The error is fixed and we can continue typing:

LOAD "B:PROGRAM1"

**Erasing Characters:** If you notice you've typed an extra character in the line you're typing, you can erase (delete) it using the Del key. Use the Cursor Left or other cursor movement keys to move the cursor to the character you want to erase. Press the Del key, and it is deleted. Then use the Cursor Right or End keys to move the cursor back to the end of the line, and continue typing.

For example, suppose we typed the following:

DEELETE\_

To erase the extra E, we press Cursor Left until the cursor is under the extra E:

DEELETE

Then we press the Del key:

DELETE

Then we press the End key:

DELETE

and continue typing:

DELETE 20

If the incorrect character was the character you just typed, use the Backspace key to delete it. Then you can simply continue typing the line as desired.

For example, suppose we've typed the following:

DELETT

We can simply press the Backspace key:

DELET

Then we can continue typing:

DELETE 20

**Adding Characters:** If you see that you've omitted characters in the line you're typing, move the cursor to the position where you want to put the new characters. Press the Ins key to get into Insert Mode. Type the characters you want to add. The characters you type will be inserted at the cursor and the characters above and following it will be pushed to the right. As before, when you're ready to continue typing at the end of the line, use the Cursor Right or End keys to move the cursor there and just continue typing. Insert Mode will automatically be turned off when you use either of these keys.

For example, suppose we've typed the following:

LIS 10\_

We forgot the T in LIST. So we press Cursor Left until the cursor is under the space:

LIS\_10

Then we press the Ins key and type the letter T:

LIST<sub>■</sub>10

**Erasing Part of a Line:** To truncate a line at the current cursor position, press Ctrl-End.

For example, suppose we have the following:

10 REM \*\*\* garbage garbage garbage

We have the cursor positioned under the g in the first word **garbage**, so all we have to do to erase the garbage is press Ctrl-End:

10 REM \*\*\* \_

**Cancelling a Line:** To cancel a line that is in the process of being typed, press the Esc key anywhere in the line. You do not have to press Enter. This causes the entire logical line to be erased.

For example, suppose we had this line:

THIS IS A LINE THAT HAS NO MEANING\_

Even though the cursor is at the end of the line, the entire line is erased when we press Esc:

\_

# Entering or Changing a BASIC Program

Any line of text that you type that begins with a number is considered to be a *program line*.

A BASIC program line always begins with a line number, ends with an Enter, and may contain a maximum of 255 characters, including the Enter. If a line contains more than 255 characters, the extra characters will be truncated when the Enter is pressed. Even though the extra characters still appear on the screen, they are not processed by BASIC.

BASIC keywords and variable names must be in uppercase. However, you may enter them in any combination of uppercase and lowercase. The program editor will convert everything to uppercase, except for remarks, DATA statements, and strings enclosed in quotation marks.

BASIC will sometimes change the way you enter something in other ways. For example, suppose you use the question mark (?) instead of the word PRINT in a program line. When you later LIST the line, the ? will be changed to PRINT with a space after it, since ? is a shorthand way of entering PRINT. This expansion may cause the end of a line to be truncated if the line length is close to 255 characters.

## Warning:

If your line reaches maximum length, the 255th character must be Enter.

**Adding a New Line to the Program:** Enter a valid line number (range is 0 through 65529) followed by at least one non-blank character, followed by Enter. The line will be saved as part of the BASIC program in storage.

For example, if you enter the following:

```
10 hello Dori
```

This will save the line as line number 10 in the program. Note that **hello Dori** is not a valid BASIC statement; however, you will not get an error if you enter this line. Program lines are *not* checked for proper syntax before being added to the program. That only happens when the program line is actually executed.

If a line already exists with the same line number, then the old line is erased and replaced with the new one.

If you try to add a line to a program when there is no more room in storage, an “Out of memory” error occurs and the line is not added.

**Replacing or Changing an Existing Program Line:** An existing line is changed, as indicated above, when the line number of the line you enter matches the line number of a line already in the program. The old line is replaced with the text of the new one.

For example, if you enter:

```
10 this is a new line 10
```

The previous line 10 ( **hello Dori** ) would be replaced with this new line 10.

**Deleting Program Lines:** To delete an existing program line, type the line number alone followed by Enter. For example, if you enter:

```
10
```

This would delete line 10 from the program.

Or you may use the DELETE command to delete a group of program lines. Refer to “DELETE Command” in Chapter 4 for details.

Note that if you try to delete a non-existent line, an “Undefined line number” error will occur.

Do not use the Esc key to delete program lines. Esc will cause the line to be erased from the screen only. If the line exists in the BASIC program, it will remain there.

**Deleting an Entire Program:** To delete the entire program that is currently residing in memory, enter the NEW command (see “NEW Command” in Chapter 4). NEW is usually used to clear memory prior to entering a new program.

## Changing Lines Anywhere on the Screen

You can edit any line on the screen simply by using the cursor movement keys (described under “Special Program Editor Keys”) to move the cursor on the screen to the place requiring the change. Then you can use any or all of the techniques described previously to change, delete, or add characters to the line.

If you want to modify program lines that do not happen to be displayed at the moment, you can use the LIST command to display them. List the line or range of lines to be edited (see “LIST Command” in Chapter 4). Position the cursor to a line to be edited and change the line using the techniques already described. Press Enter to store the modified line in the program. You can also use the EDIT command to display the line you want. Refer to “EDIT Command” in Chapter 4.

For example, you could duplicate a line in the program this way: Move the cursor to the line to be duplicated. Change the line number to the new line number by just typing over the numbers. When you press Enter, both the old line and the new line will be in the program.

Or, you could change the line number of a program line by duplicating the line as described above, then deleting the old line.

A program line is never actually changed within the BASIC program until Enter is pressed. Therefore, when several lines need alteration, it may be easier to move around the screen making corrections to several lines at once, and then go back to the first line changed and press Enter at the beginning of each line. By so doing, you store each modified line in the program.

You do not have to move the cursor to the end of the logical line before pressing Enter. The program editor knows where each logical line ends and it processes the whole line even if the Enter is pressed at the beginning of the line.

**Note:** Use of the AUTO command can be very helpful when you are entering your program. However, you should exit AUTO mode by pressing Ctrl-Break before changing any lines other than the current one.

Remember, changes made using these techniques only change the program in memory. To save the program with the new changes permanently, you should use the SAVE command (see "SAVE Command" in Chapter 4) before entering a NEW command or leaving BASIC.

## Syntax Errors

When a syntax error is discovered while a program is running, BASIC automatically displays the line that caused the error so you may correct it. For example:

```
Ok  
1Ø A = 2$12  
RUN  
Syntax error in 1Ø  
Ok  
1Ø A = 2$12
```

The program editor has displayed the line in error and positioned the cursor right under the digit 1. You can move the cursor right to the dollar sign (\$) and change it to a plus sign (+), then press Enter. The corrected line is now stored back in the program.

When you edit a line and store it back in the program while the program is interrupted (as in this example) certain things happen, primarily:

- All variables and arrays are lost. That is, they are reset to zero or null.
- Any files that were open are closed.
- You cannot use CONT to continue the program.

If you want to examine the contents of some variable before making the change, you should press Ctrl-Break to return to command level. The variables will be preserved since no program line is changed. After you check everything you need to, you can edit the line and rerun the program.

# CHAPTER 3. GENERAL INFORMATION ABOUT PROGRAMMING IN BASIC

## Contents

Line Format .....	3-3
Line Numbers .....	3-3
BASIC Statements .....	3-3
Comments .....	3-4
Character Set .....	3-4
Reserved Words .....	3-6
Constants .....	3-9
Numeric Precision .....	3-11
Variables .....	3-12
How to Name a Variable .....	3-12
How to Declare Variable Types .....	3-13
Arrays .....	3-15
How BASIC Converts Numbers from One Precision to Another .....	3-18
Numeric Expressions and Operators .....	3-21
Arithmetic Operators .....	3-21
Integer Division .....	3-22
Modulo Arithmetic .....	3-22
Relational Operators .....	3-23
Numeric Comparisons .....	3-23
String Comparisons .....	3-24
Logical Operators .....	3-25
How Logical Operators Work .....	3-27
Numeric Functions .....	3-29
Order of Execution .....	3-29

GENERAL INFO

<b>String Expressions and Operators .....</b>	<b>3-31</b>
Concatenation .....	3-31
String Functions .....	3-32
 <b>Input and Output .....</b>	 <b>3-33</b>
Files .....	3-33
Naming Files .....	3-34
Using the Screen .....	3-38
Display Adapters .....	3-38
Text Mode .....	3-39
Graphics Modes .....	3-41
Other I/O Features .....	3-44
Clock .....	3-44
Sound and Music .....	3-44
Light Pen .....	3-45
Joysticks .....	3-45

# Line Format

Program lines in a BASIC program have the following format:

nnnnn BASIC statement[:BASIC statement...][‘ comment ]

and they end with Enter. This format is explained in more detail below.

**Line Numbers:** “nnnnn” indicates the line number, which can be from one to five digits. Every BASIC program line begins with a line number. Line numbers are used to show the order in which the program lines are stored in memory and also as reference points for branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in LIST, AUTO, DELETE, and EDIT commands to refer to the current line.

**BASIC Statements:** A BASIC statement is either *executable* or *non-executable*. Executable statements are program instructions that tell BASIC what to do next while running a program. For example, PRINT X is an executable statement. Non-executable statements, such as DATA or REM, do not cause any program action when BASIC sees them. All the BASIC statements are explained in detail in the next chapter.

You may, if you wish, have more than one BASIC statement on a line, but each statement on a line must be separated from the last one by a colon, and the total number of characters must not exceed 255.

For example:

```
Ok  
10 FOR I=1 TO 5: PRINT I: NEXT  
RUN  
1  
2  
3  
4  
5  
Ok
```

**Comments:** Comments may be added to the end of a line using the ' (single quote) to separate the comment from the rest of the line.

## Character Set

The BASIC character set consists of alphabetic characters, numeric characters and special characters. These are the characters which BASIC recognizes.

The alphabetic characters in BASIC are the uppercase and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9.

The following special characters have specific meanings in BASIC:

<i>Character</i>	<i>Name</i>
	blank
=	equal sign or assignment symbol
+	plus sign or concatenation symbol
-	minus sign
*	asterisk or multiplication symbol
/	slash or division symbol
\	backslash or integer division symbol
^	caret or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent sign or integer type declaration character
#	number (or pound) sign, or double-precision type declaration character
\$	dollar sign or string type declaration character
!	exclamation point or single-precision type declaration character
&	ampersand
,	comma
:	period or decimal point
'	single quotation mark (apostrophe), or remark delimiter
;	semicolon
:	colon or statement separator
?	question mark (PRINT abbreviation)
<	less than
>	greater than
"	double quotation mark or string delimiter
_	underline

Many characters can be printed or displayed even though they have no particular meaning to BASIC. See "Appendix G. ASCII Character Codes" for a complete list of these characters.

# Reserved Words

Certain words have special meaning to BASIC. These words are called *reserved words*. Reserved words include all BASIC commands, statements, function names, and operator names. Reserved words cannot be used as variable names.

You should always separate reserved words from data or other parts of a BASIC statement by using spaces or other special characters as allowed by the syntax. That is, the reserved words must be appropriately *delimited* so that BASIC will recognize them.

The following is a list of all the reserved words in BASIC.

ABS	CVD
AND	CVI
ASC	CVS
ATN	DATA
AUTO	DATE\$
BEEP	DEF
BLOAD	DEFDBL
BSAVE	DEFINT
CALL	DEFSNG
CDBL	DEFSTR
CHAIN	DELETE
CHR\$	DIM
CINT	DRAW
CIRCLE	EDIT
CLEAR	ELSE
CLOSE	END
CLS	EOF
COLOR	EQV
COM	ERASE
COMMON	ERL
CONT	ERR
COS	ERROR
CSNG	EXP
CSRLIN	FIELD

**GENERAL INFO**

FILES	NOT
FIX	OCT\$
FNxxxxxxxx	OFF
FOR	ON
FRE	OPEN
GET	OPTION
GOSUB	OR
GOTO	OUT
HEX\$	PAINT
IF	PEEK
IMP	PEN
INKEY\$	PLAY
INP	POINT
INPUT	POKE
INPUT#	POS
INPUT\$	PRESET
INSTR	PRINT
INT	PRINT#
KEY	PSET
KILL	PUT
LEFT\$	RANDOMIZE
LEN	READ
LET	REM
LINE	RENUM
LIST	RESET
LLIST	RESTORE
LOAD	RESUME
LOC	RETURN
LOCATE	RIGHT\$
LOF	RND
LOG	RSET
LPOS	RUN
LPRINT	SAVE
LSET	SCREEN
MERGE	SGN
MID\$	SIN
MKD\$	SOUND
MKI\$	SPACE\$
MKS\$	SPC(
MOD	SQR
MOTOR	STEP
NAME	STICK
NEW	STOP
NEXT	STR\$

STRIG	USR
STRING\$	VAL
SWAP	VARPTR
SYSTEM	VARPTR\$
TAB(	WAIT
TAN	WEND
THEN	WHILE
TIME\$	WIDTH
TO	WRITE
TROFF	WRITE#
TRON	XOR
USING	

# Constants

Constants are the actual values BASIC uses during execution. There are two types of constants: string (or character) constants, and numeric constants.

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

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

Numeric constants are positive or negative numbers. A plus sign (+) is optional on a positive number. Numeric constants in BASIC cannot contain commas. There are five ways to indicate numeric constants:

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

**Fixed point** Positive or negative real numbers, that is, numbers that contain decimal points.

**Floating point** Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). Double-precision floating point constants use the letter D instead of E. For more information, see the next section, "Numeric Precision."

The E (or D) means “times ten to the power of.”

For example,

23E-2

Here, 23 is the mantissa, and -2 is the exponent. This number could be read as “twenty-three times ten to the negative two power.” You could write it as 0.23 in regular fixed point notation. More examples:

235.988E-7

is equivalent to: .0000235988

2359D6

is equivalent to: 2359000000

You can represent any number from 2.9E-39 to 1.7E+38 (positive or negative) as a floating point constant.

### Hex

Hexadecimal numbers with up to four digits, with a prefix of &H. Hexadecimal digits are the numbers 0 through 9, A, B, C, D, E, and F. Examples:

&H76  
&H32F

### Octal

Octal numbers with up to 6 digits, with the prefix &O or just &. Octal digits are 0 through 7. Examples:

&0349  
&1234

# Numeric Precision

Numbers may be stored internally as either integer, single-precision, or double-precision numbers.

Constants entered in integer, hex, or octal format are stored in two bytes of memory and are interpreted as integers or whole numbers. With double-precision, the numbers are stored with 17 digits of precision and printed with up to 16 digits. With single-precision, seven digits are stored and up to seven digits are printed, although only six digits will be accurate.

A single-precision constant is any numeric constant that doesn't fit in the *integer* category and is written with:

- seven or fewer digits, or
- exponential form using E, or
- a trailing exclamation point (!)

A double-precision constant is any numeric constant that is written with:

- eight or more digits, or
- exponential form using D, or
- a trailing number sign (#)

Examples of single- and double-precision constants:

## Single-Precision

46.8  
-1.09E-06  
3489.0  
22.5!

## Double-Precision

345692811  
-1.09432D-06  
3489.0#  
7654321.1234

# Variables

Variables are names used to represent values that are used in a BASIC program. As with constants, there are two types of variables: numeric and string. A numeric variable always has a value that is a number. A string variable may only have a character string value.

The length of a string variable is not fixed, but may be anywhere from 0 (zero) to 255 characters. The length of the string value you assign to it will determine the length of the variable.

You may set the value of a variable to a constant, or you can set its value as the result of calculations or various data input statements in the program. In either case, the variable type (string or numeric) must match the type of data that is being assigned to it.

If you use a numeric variable before you assign a value to it, its value is assumed to be zero. String variables are initially assumed to be null; that is, they have no characters in them and have a length of zero.

## How to Name a Variable

BASIC variable names may be any length. If the name is longer than 40 characters, however, only the first 40 characters are significant.

The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter. Special characters which identify the type of variable are also allowed as the last character of the name. For more information about types, see the next section, "How to Declare Variable Types."

A variable name may not be a reserved word, but may contain imbedded reserved words. (Refer to "Reserved Words," earlier in this chapter, for a complete list of the reserved words.) Also, a variable name may not be a reserved word with one of the type declaration characters (\$, %, !, #) at the end. For example,

`1Ø EXP = 5`

is invalid, because EXP is a reserved word. However,

`1Ø EXPONENT = 5`

is okay, because EXP is only a part of the variable name.

A variable beginning with FN is assumed to be a call to a user-defined function (see "DEF FN Statement" in Chapter 4).

## How to Declare Variable Types

A variable's name determines its type (string or numeric, and if numeric, what its precision is).

String variable names are written with a dollar sign (\$) as the last character. For example:

`A$ = "SALES REPORT"`

The dollar sign is a variable type declaration character. It "declares" that the variable will represent a string.

Numeric variable names may declare integer, single-, or double-precision values. Although you may get less accuracy doing computations with integer and single-precision variables, there are

reasons you might want to declare a variable as a particular precision.

- Variables of higher precisions take up more room in storage. This is important if space is a consideration.
- It takes more time for the computer to do arithmetic with the higher precision numbers. A program with repeated calculations will run faster with integer variables.

The type declaration characters for numeric variables and the number of bytes required to store each type of value are as follows:

- % Integer variable (2 bytes)
- ! Single-precision variable (4 bytes)
- # Double-precision variable (8 bytes)

If the variable type is not explicitly declared, then it will default to single-precision.

Examples of BASIC variable names follow.

PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
LIMIT%	declares an integer value
N\$	declares a string value
ABC	represents a single-precision value

Variable types may also be declared in another way. The BASIC statements DEFINT, DEFSNG, DEFDBL and DEFSTR may be included in a program to declare the types for certain variable names. These statements are described in detail under “DEFtype Statements” in Chapter 4. All the examples which follow in this book assume that none of these types of declarations have been made, unless the statements are explicitly shown in the example.

## Arrays

An array is a group or table of values that are referred to with one name. Each individual value in the array is called an *element*. Array elements are variables and can be used in expressions and in any BASIC statement or function which uses variables.

Declaring the name and type of an array and setting the number of elements and their arrangement in the array is known as *defining*, or *dimensioning*, the array. Usually this is done using the DIM statement. For example,

```
10 DIM B$(5)
```

This creates a one dimensional array named B\$. All its elements are variable length strings, and the elements have an initial null value.

```
20 DIM A(2,3)
```

This creates a two-dimensional array named A. Since the name does not have a type declaration character, the array consists of single-precision values. All the array elements are initially set to 0.

Each array element is named with the array name *subscripted* with a number or numbers. An array variable name has as many subscripts as there are dimensions in the array.

The subscript indicates the position of the element in the array. Zero is the lowest position unless you explicitly change it (see “OPTION BASE Statement” in Chapter 4). The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

To continue the preceding examples, array B\$ could be thought of as a list of character strings, like this:

B\$(0)
B\$(1)
B\$(2)
B\$(3)
B\$(4)
B\$(5)

The first string in the list is named B\$(0).

The array A could be thought of as a table of rows and columns, like this:

		columns			
		A(0,0)	A(0,1)	A(0,2)	A(0,3)
rows	A(1,0)	A(1,1)	A(1,2)	A(1,3)	
	A(2,0)	A(2,1)	A(2,2)	A(2,3)	

The element in the second row, first column, is called A(1,0).

If you use an array element before you define the array, it is assumed to be dimensioned with a maximum subscript of 10.

For example, if BASIC encounters the statement:

50 SIS(3)=500

and the array SIS has not already been defined, the array is set to a one-dimensional array with 11 elements, numbered SIS(0) through SIS(10). You may only use this method of *implicit declaration* for one-dimensional arrays.

One final example:

```
0k
10 DIM YEARS(3,4)
20 YEARS(2,3)=1982
30 FOR ROW=0 TO 3
40 FOR COLUMN=0 TO 4
50 PRINT YEARS(ROW,COLUMN);
60 NEXT COLUMN
70 PRINT
80 NEXT ROW
RUN
    0   0   0   0   0
    0   0   0   0   0
    0   0   0   1982   0
    0   0   0   0   0
0k
```

# How BASIC Converts Numbers from One Precision to Another

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

1. If a numeric value of one precision is assigned to a numeric variable of a different precision, the number will be stored as the precision declared in the target variable name.

Example:

```
0k  
1Ø A% = 23.42  
2Ø PRINT A%  
RUN  
23  
0k
```

2. Rounding, as opposed to truncation, occurs when assigning any higher precision value to a lower precision variable (for example, changing from double- to single-precision).

Example:

```
0k  
1Ø C = 55.8834567#  
2Ø PRINT C  
RUN  
55.88346  
0k
```

This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 0 for X).

3. If you convert from a lower precision to a higher precision number, the resulting higher precision number cannot be any more accurate than the lower precision number. For example, if you assign a single-precision value (A) to a double-precision variable (B#), only the first six digits of B# will be accurate because only six digits of accuracy were supplied with A. The error can be bounded using the following formula:

$$\text{ABS}(B\# - A) < 6.3E-8 * A$$

That is, the absolute value of the difference between the printed double-precision number and the original single-precision value is less than 6.3E-8 times the original single-precision value.

Example:

```
0k
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
2.04  2.039999961853027
0k
```

- When an expression is evaluated, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, namely that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Examples:

```
0k  
1Ø D# = 6#/7  
2Ø PRINT D#  
RUN  
.8571428571428571  
0k
```

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

```
0k  
1Ø D = 6#/7  
2Ø PRINT D  
RUN  
.8571429  
0k
```

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

- Logical operators (see “Logical Operators” in this chapter) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an “Overflow” error occurs.

# Numeric Expressions and Operators

A numeric expression may be simply a numeric constant or variable. It may also be used to combine constants and variables using operators to produce a single numeric value.

Numeric operators perform mathematical or logical operations mostly on numeric values, and sometimes on string values. We refer to them as “numeric” operators because they produce a value that is a number. The BASIC numeric operators may be divided into categories as follows:

- Arithmetic
- Relational
- Logical
- Functions

## Arithmetic Operators

The arithmetic operators perform the usual operations of arithmetic, such as addition and subtraction. In order of precedence, they are:

Operator	Operation	Sample Expression
$^$	Exponentiation	$X^Y$
$-$	Negation	$-X$
$\ast, /$	Multiplication, Floating Point Division	$X^*Y$ $X/Y$
$\backslash$	Integer Division	$X\backslash Y$
MOD	Modulo Arithmetic	$X \text{ MOD } Y$
$+, -$	Addition, Subtraction	$X+Y$ $X-Y$

(If you have a mathematical background, you will notice that this is the standard order of precedence.) Although most of these operations probably look familiar to you, two of them may seem a bit unfamiliar — integer division and modulo arithmetic.

**Integer Division:** Integer division is denoted by the backslash (\). The operands are rounded to integers (in the range -32768 to 32767) before the division is performed; the quotient is truncated to an integer.

For example:

```
0k  
10 A = 10\4  
20 B = 25.68\6.99  
30 PRINT A;B  
RUN  
    2   3  
0k
```

**Modulo Arithmetic:** Modulo arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division.

For example:

```
0k  
10 A = 7 MOD 4  
20 PRINT A  
RUN  
    3  
0k
```

This result occurs because  $7/4$  is 1, with remainder 3.

```
0k  
PRINT 25.68 MOD 6.99  
5  
0k
```

The result is 5 because  $26/7$  is 3, with the remainder 5. (Remember, BASIC rounds when converting to integers.)

# Relational Operators

Relational operators compare two values. The values may be either both numeric, or both string. The result of the comparison is either "true" (-1) or "false" (0). This result is usually then used to make a decision regarding program flow. (See "IF Statement" in Chapter 4.)

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

(The equal sign is also used to assign a value to a variable. See "LET Statement" in Chapter 4.)

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

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

will be true (-1) if the value of X plus Y is less than the value of T-1 divided by Z.

More examples:

```
Ok  
1Ø X=1ØØ  
2Ø IF X <> 2ØØ THEN PRINT "NOT EQUAL"  
                      ELSE PRINT "EQUAL"  
RUN  
NOT EQUAL  
Ok
```

Here, the relation is true (100 is not equal to 200). The true result causes the THEN part of the IF statement to be executed.

```
Ok  
PRINT 5<2; 5<1Ø  
      0 -1  
Ok
```

Here the first result is false (zero) because 5 is not less than 2. The second result is -1 because the expression  $5 < 10$  is true.

**String Comparisons:** String comparisons can be thought of as “alphabetical.” That is, one string is “less than” another if the first string comes before the other one alphabetically. Lowercase letters are “greater than” their uppercase counterparts. Numbers are “less than” letters.

The way two strings are actually compared is by taking one character at a time from each string and comparing the ASCII codes. (See “Appendix G. ASCII Character Codes” for a complete list of ASCII codes.) If all the ASCII codes are the same, the strings are equal. Otherwise, as soon as the ASCII codes differ, the string with the lower code number is less than the string with the higher code number. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller.

Leading and trailing blanks are significant. For example, all the following relational expressions are true (that is, the result of the relational operation is -1).

```
"AA" < "AB"  
"FILENAME" = "FILENAME"  
"X&" > "X#"  
"WR" > "WR"  
"kg" > "KG"  
"SMYTH" < "SMYTHE"  
B$ < "718"           (where B$ = "12543")
```

All string constants used in comparison expressions must be enclosed in quotation marks.

## Logical Operators

Logical operators perform logical, or *Boolean*, operations on numeric values. Just as the relational operators are usually used to make decisions regarding program flow, logical operators are usually used to connect two or more relations and return a true or false value to be used in a decision (see “IF Statement” in Chapter 4).

A logical operator takes a combination of true-false values and returns a true or false result. An operand of a logical operator is considered to be “true” if it is not equal to zero (like the -1 returned by a relational operator), or “false” if it is equal to zero. The result of the logical operation is a number which is, again, “true” if it is not equal to zero, or “false” if it is equal to zero. The number is calculated by performing the operation bit by bit. This is explained in detail shortly.

The logical operators are NOT (logical complement), AND (conjunction), OR

(disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in the following table. ("T" indicates a true, or non-zero value. "F" indicates a false, or zero value.) The operators are listed in order of precedence.

### NOT

X	NOT X
T	F
F	T

### AND

X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F

### OR

X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F

### XOR

X	Y	X XOR Y
T	T	F
T	F	T
F	T	T
F	F	F

### EQV

X	Y	X EQV Y
T	T	T
T	F	F
F	T	F
F	F	T

### IMP

X	Y	X IMP Y
T	T	T
T	F	F
F	T	T
F	F	T

Some examples of ways to use logical operators in decisions:

```
IF HE>60 AND SHE<20 THEN 1000
```

Here, the result will be true if the value of the variable HE is more than 60 and also the value of SHE is less than 20.

```
IF I>10 OR K<0 THEN 50
```

The result will be true if I is greater than 10, or K is less than 0, or both.

```
50 IF NOT (P=-1) THEN 100
```

Here, the program will branch to line 100 if P is not equal to -1. Note that NOT (P=-1) does not produce the same result as NOT P. Refer to the next section, "How Logical Operators Work," for an explanation.

```
100 FLAG% = NOT FLAG%
```

This example switches a value back and forth from true to false.

**How Logical Operators Work:** Operands are converted to integers in the range -32768 to +32767. (If the operands are not in this range, an "Overflow" error results.) If the operand is negative, the two's complement form is used. This turns each operand into a sequence of 16 bits. The operation is performed on these sequences. That is, each bit of the result is determined by the corresponding bits in the two operands, according to the tables for the operator listed previously. A 1 bit is considered "true", and a 0 bit is "false."

Thus, you can use logical operators to test for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port.

The following examples will help demonstrate how the logical operators work.

A = 63 AND 16

Here, A is set to 16. Since 63 is binary 111111 and 16 is binary 10000, 63 AND 16 equals 010000 in binary, which is equal to 16.

B = -1 AND 8

B is set to 8. Since -1 is binary 11111111 11111111 and 8 is binary 1000, -1 AND 8 equals binary 00000000 00001000, or 8.

C = 4 OR 2

Here, C equals 6. Since 4 is binary 100 and 2 is binary 010, 4 OR 2 is binary 110, which is equal to 6.

X = 2

TWOSCOMP = (NOT X) + 1

This example shows how to form the two's complement of a number. X is 2, which is 10 binary. NOT X is then binary 11111111 11111101, which is -3 in decimal; -3 plus 1 is -2, the complement of 2. That is, the two's complement of any integer is the bit complement plus one.

Note that if both operands are equal to either 0 or -1, a logical operator will return either 0 or -1.

# Numeric Functions

A function is used like a variable in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has “built-in” functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC’s built-in functions are listed under “Functions and Variables” in the beginning of Chapter 4. Detailed descriptions are also included in the alphabetical section of Chapter 4.

You can also define your own functions using the DEF FN statement. See “DEF FN Statement” in Chapter 4.

## Order of Execution

The categories of numeric operations were discussed in their order of precedence, and the precedence of each operation within a category was indicated in the discussion of the category. In summary:

1. Function calls are evaluated first
2. Arithmetic operations are performed next, in this order:
  - a. ^
  - b. unary -
  - c. \*, /
  - d. \
  - e. MOD
  - f. +, -
3. Relational operations are done next

4. Logical operations are done last, in this order:

- a. NOT
- b. AND
- c. OR
- d. XOR
- e. EQV
- f. IMP

Operations at the same level in the list are performed in left-to-right order. To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
----------------------	------------------

$$X+2Y$$

$$X+Y^*2$$

$$\frac{X-Y}{Z}$$

$$X-Y/Z$$

$$\frac{XY}{Z}$$

$$X^*Y/Z$$

$$\frac{X+Y}{Z}$$

$$(X+Y)/Z$$

$$(X^2)^Y$$

$$(X^2)^Y$$

$$X^{Y^Z}$$

$$X^(Y^Z)$$

$$X(-Y)$$

$$X*(-Y)$$

**Note:** Two consecutive operators must be separated by parentheses, as shown in the  $X^*(-Y)$  example.

# String Expressions and Operators

A string expression may be simply a string constant or variable, or it may combine constants and variables by using operators to produce a single string value.

String operators are used to arrange character strings in different ways. The two categories of string operators are:

- Concatenation
- Functions

Note that although you can use the relational operators =, <>, <, >, <=, and >= to compare two strings, these are not considered to be “string operators” because they produce a numeric result, not a string result. Read through “Relational Operators” earlier in this chapter for an explanation of how you can compare strings using relational operators.

## Concatenation

Joining two strings together is called *concatenation*. Strings are concatenated using the plus symbol (+). For example:

```
0k
10 COMPANY$ = "IBM"
20 TYPE$ = " Personal"
30 FULLNAME$ = TYPE$ + " Computer"
40 PRINT COMPANY$+FULLNAME$
RUN
IBM Personal Computer
0k
```

## String Functions

A string function is like a numeric function except that it returns a string result. A string function can be used in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has “built-in” functions that reside in the system, such as MID\$, which returns a string from the middle of another string, or CHR\$, which returns the character with the specified ASCII code. All of BASIC’s built-in functions are listed under “Functions and Variables” in the beginning of Chapter 4. Detailed descriptions are also included in the alphabetical section of Chapter 4.

You can also define your own functions using the DEF FN statement. See “DEF FN Statement” in Chapter 4.

# Input and Output

The remainder of this chapter contains information on input and output (I/O) in BASIC. The following topics are addressed:

- Files — how BASIC uses files, how to name files, and device names
- The screen — ways to display things on the screen, with emphasis on graphics
- Other features — clock, sound, light pen, and joysticks

## Files

A file is a collection of information which is kept somewhere other than in the random access memory of the IBM Personal Computer. For example, your information may be stored in a file on diskette or cassette. In order to use the information, you must *open* the file to tell BASIC where the information is. Then you may use the file for input and/or output.

BASIC supports the concept of general device I/O files. This means that any type of input/output may be treated like I/O to a file, whether you are actually using a cassette or diskette file, or are communicating with another computer.

**File Number:** BASIC performs I/O operations using a file number. The file number is a unique number that is associated with the actual physical file when it is opened. It identifies the path for the collection of data. A file number may be any number, variable, or expression ranging from 1 to  $n$ , where  $n$  is the maximum number of files allowed.  $n$  is 4 in Cassette BASIC, and defaults to 3 in Disk and Advanced BASIC. It may be changed, up to a maximum of 15, by using the /F: option on the BASIC command for Disk and Advanced BASIC.

## Naming Files

The physical file is described by its *file specification*, or *filespec* for short.

The file specification is a string expression of the form:

*device:filename*

The device name tells BASIC *where* to look for the file, and the filename tells BASIC *which* file to look for on that particular device. Sometimes you do not need both device name and filename, so specification of device and filename is optional.

Note the colon (:) indicated above. Whenever you specify a device, you must use the colon even though a filename is not necessarily specified. From now on we will include the colon as part of the device name.

**Note:** File specification for communications devices is slightly different. The filename is replaced with a list of options specifying such things as line speed. Refer to “OPEN “COM... Statement” in Chapter 4 for details.

Remember that if you use a string constant for the *filespec*, you must enclose it in quotation marks. For example,

```
LOAD "B:ROTERM.ARK"
```

**Device Name:** The device name consists of up to four characters followed by a colon (:). The following is a complete list of device names, telling what device the name applies to, what the device can be used for (input or output), and which versions of BASIC support the device.

## Device Name Chart

- KYBD: Keyboard. Input only, all versions of BASIC.
- SCRN: Screen. Output only, all versions of BASIC.
- LPT1: First printer. Output, all versions; or random, Disk and Advanced BASIC.
- LPT2: Second printer. Output or random, Disk and Advanced BASIC.
- LPT3: Third printer. Output or random, Disk and Advanced BASIC.

## COMMUNICATIONS DEVICES

- COM1: First Asynchronous Communications Adapter. Input and output, Disk and Advanced BASIC.
- COM2: Second Asynchronous Communications Adapter. Input and output, Disk and Advanced BASIC.

## STORAGE DEVICES

- CAS1: Cassette tape player. Input and output, all versions.
- A: First diskette drive. Input and output, Disk and Advanced BASIC.
- B: Second diskette drive. Input and output, Disk and Advanced BASIC.

Refer to "Search Order for Adapters" in "Appendix I. Technical Information and Tips" for information on which adapters are referred to by the printer and communications device names.

**Filename:** The filename must conform to the following rules.

For cassette files:

- The name may not be more than eight characters long.
- The name may not contain colons, hex '00's or hex 'FF's (decimal 255s).

For diskette files, the name should conform to DOS conventions:

- The name may consist of two parts separated by a period ( . ):

*name.extension*

The *name* may be from one to eight characters long. The *extension* may be no more than three characters long.

If *extension* is longer than three characters, the extra characters are truncated. If *name* is longer than eight characters and *extension* is not included, then BASIC inserts a period after the eighth character and uses the extra characters (up to three) for the *extension*. If *name* is longer than eight characters and an *extension* is included, then an error occurs.

- Only the following characters are allowed in *name* and *extension*:

A through Z  
0 through 9  
< > ( ) { }  
@ # \$ % ^ & !  
\_ \_ \ ~ |

Some examples of filenames for Disk and Advanced BASIC are:

27HAL.DAD

VDL

PROGRAM1.BAS

\$\$@(!).123

The following examples show how BASIC truncates names and extensions when they are too long, as explained above.

A23456789JKLMN becomes: A2345678.9JK

@HOME.TRUM10 becomes: @HOME.TRU

SHERRYLYNN.BAS causes an error

# Using the Screen

BASIC can display text, special characters, points, lines, or more complex shapes in color or in black and white. How much of this you can do depends on which display adapter you have in your IBM Personal Computer.

## Display Adapters

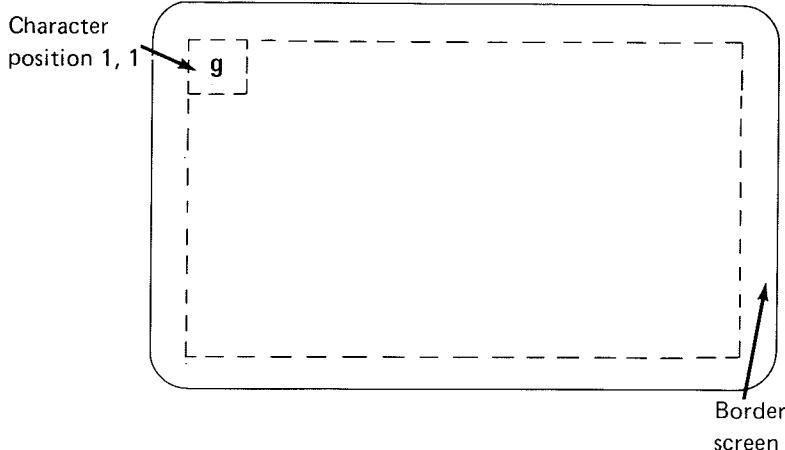
The IBM Personal Computer has two display adapters: the IBM Monochrome Display and Parallel Printer Adapter, and the Color/Graphics Monitor Adapter.

With the IBM Monochrome Display and Parallel Printer Adapter, you can display text in black and white. *Text* refers to letters, numbers, and all the special characters in the regular character set. You have some capability to draw pictures with the special line and block characters. You can also create blinking, reverse image, invisible, highlighted, and underscored characters by setting parameters on the COLOR statement.

The Color/Graphics Monitor Adapter also operates in text mode, but it allows you to display text in 16 different colors. (You can also display in just black and white by setting parameters on the SCREEN or COLOR statements.) You also get complete graphics capability to draw complex pictures. This graphics capability makes the screen *all points addressable* in medium and high resolution. This is more versatile than the ability to draw with the special line and block characters which you have in text mode. From now on, the term *graphics* will refer only to this special capability of the Color/Graphics Monitor Adapter. The use of the extended character set with special line and block characters will not be considered graphics.

## Text Mode

The screen can be pictured like this:



Characters are shown in 25 horizontal lines across the screen. These lines are numbered 1 through 25, from top to bottom. Each line has 40 character positions (or 80, depending on how you set the width). These are numbered 1 to 40 (or 80) from left to right. The position numbers are used by the LOCATE statement, and are the values returned by the POS(0) and CSRLIN functions. For example, the character in the upper left corner of the screen is on line 1, position 1.

Characters are normally placed on the screen using the PRINT statement. The characters are displayed at the position of the cursor. Characters are displayed from left to right on each line, from line 1 to line 24. When the cursor would normally go to line 25 on the screen, lines 1 through 24 are *scrolled* up one line, so that what was line 1 disappears from the screen. Line 24 is then blank, and the cursor remains on line 24 to continue printing.

Line 25 is usually used for "soft key" display (see "KEY Statement" in Chapter 4), but it is possible to write over this area of the screen if you turn the "soft key" display off. The 25th line is never scrolled by BASIC.

Each character on the screen is composed of two parts: foreground and background. The foreground is the character itself. The background is the "box" around the character. You can set the foreground and the background color for each character using the COLOR statement. You can also choose to make characters blink.

You can use a total of 16 different colors if you have the Color/Graphics Monitor Adapter:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

The colors may vary depending on your particular display device. Adjusting the color tuning of the display may help get the colors to match this chart better.

Most television sets or monitors have an area of "overscan" which is outside the area used for characters. This overscan area is known as the *border screen*. You can also use the COLOR statement to set the color of the border screen.

The statements you can use to display information in text mode are:

CLS	SCREEN
COLOR	WIDTH
LOCATE	WRITE
PRINT	

The following functions and system variables may be used in text mode:

CSRLIN	SPC
POS	TAB
SCREEN	

Another special feature you get in text mode if you have the Color/Graphics Monitor Adapter is multiple display pages. The Color/Graphics Monitor Adapter has a 16K-byte screen buffer, but text mode needs only 2K of that (or 4K for 80 column width). So the buffer is divided into different *pages*, which can be written on and/or displayed individually. There are 8 pages, numbered 0 to 7, in 40 column width; and 4 pages, numbered 0 to 3, in 80 column width. Refer to "SCREEN Statement" in Chapter 4 for details.

## Graphics Modes

The graphics modes are available only if you have the Color/Graphics Monitor Adapter.

You can use BASIC statements to draw in two graphic resolutions:

- medium resolution: 320 by 200 points and 4 colors
- high resolution: 640 by 200 points and 2 colors

You can select which resolution you want to use with the SCREEN statement.

The statements used for graphics in BASIC are:

CIRCLE	PAINT
COLOR	PRESET
DRAW	PSET
GET	PUT
LINE	SCREEN

The only graphics function is:

POINT

**Medium Resolution:** There are 320 horizontal points and 200 vertical points in medium resolution. These points are numbered from left to right and from top to bottom, starting with zero. That makes the upper left corner of the screen point (0,0), and the lower right corner point (319,199). (If you are familiar with the usual mathematical method for numbering coordinates, this may seem upside-down to you.)

Medium resolution is unusual because of its color features. When you put something on the screen in medium resolution, you can specify a color number of 0, 1, 2, or 3. These colors are not fixed, as are the 16 colors in text mode. You select the actual color for color number 0 and select one of two "palettes" for the other three colors by using the COLOR statement. A palette is a set of three actual colors to be associated with the color numbers 1, 2 and 3. If you change the palette with a COLOR statement, all the colors on the screen change to match the new palette.

You can still display text characters on the screen when you are in graphics mode. The size of the characters will be the same as in text mode; that is, 25 lines of 40 characters. In medium resolution, the foreground will be color number 3, and the background will be color number 0.

**High Resolution:** In high resolution there are 640 horizontal points and 200 vertical points. As in medium resolution, these points are numbered starting with zero so that the lower right corner point is (639,199).

High resolution is a little easier to describe than medium resolution since there are only two colors: black and white. Black is always 0 (zero), and white is always 1 (one).

When you display text characters in high resolution, you get 80 characters on a line. The foreground color is 1 (one) and the background color is 0 (zero). So characters will always be white on black.

**Specifying Coordinates:** The graphic statements require information about where on the screen you want to draw. You give this information in the form of coordinates. Coordinates are generally in the form  $(x,y)$ , where  $x$  is the horizontal position, and  $y$  is the vertical position. This form is known as *absolute form*, and refers to the actual coordinates of the point on the screen, without regard to the last point referenced.

There is another way to indicate coordinates, known as *relative form*. Using this form you tell BASIC where the point is relative to the last point referenced. This form looks like:

STEP  $(x\text{offset},y\text{offset})$

You indicate inside the parentheses the *offset* in the horizontal and vertical directions from the last point referenced.

The “last point referenced” is set by each graphics statement. When we discuss these statements in

"Chapter 4. BASIC Commands, Statements, Functions, and Variables," we will indicate what each statement sets as the last point referenced.

**Note:** Be careful about drawing beyond the limits of the screen with any graphics statement; it may confuse the last point referenced.

This example shows the use of both forms of coordinates:

```
100 SCREEN 1  
110 PSET (200,100) 'absolute form  
120 PSET STEP (10,-20) 'relative form
```

This sets two points on the screen. Their actual coordinates are (200,100) and (210,80).

## Other I/O Features

### Clock

You may set and read the following system variables:

**DATE\$** Ten-character string which is the system date, in the form *mm-dd-yyyy*.

**TIME\$** Eight-character string which indicates the time as *hh:mm:ss*.

### Sound and Music

You can use the following statements to create sound on the IBM Personal Computer:

**BEEP** Beeps the speaker.

**SOUND** Makes a single sound of given frequency and duration.

**PLAY** Plays music as indicated by a character string.

## Light Pen

BASIC has the following statements and functions to allow input from a light pen.

PEN	Function which tells whether or not the pen was triggered and gives its coordinates.
PEN	Statement which enables/disables light pen function.
ON PEN	Statement to trap light pen activity.

## Joysticks

Joysticks can be useful in an interactive environment. BASIC supports two 2-dimensional (x and y coordinate) joysticks, or four one-dimensional paddles, each of which has a button. (Four buttons are supported only in Advanced BASIC.) The following statements and functions are used for joysticks:

STICK	Function which gives the coordinates of the joystick.
STRIG	Function which gives the status of the joystick button (up or down).
STRIG	Statement which enables/disables STRIG function.
ON STRIG	Statement used to trap the button being pressed.
STRIG(n)	Statement which enables/disables the joystick button interrupt.

**Note:** The light pen may only be used if you have a Color/Graphics Monitor Adapter.

Joysticks may only be used if you have a Game Control Adapter.

# **NOTES**

# CHAPTER 4. BASIC COMMANDS, STATEMENTS, FUNCTIONS, AND VARIABLES

## Contents

How to Use This Chapter .....	4-3
Commands .....	4-6
Statements .....	4-8
Non-I/O Statements .....	4-8
I/O Statements .....	4-13
Functions and Variables .....	4-17
Numeric Functions .....	4-17
Arithmetic .....	4-17
String-Related .....	4-18
I/O and Miscellaneous .....	4-19
String Functions .....	4-21
General .....	4-21
I/O and Miscellaneous .....	4-21

### Alphabetical Listing of Commands, Statements, Functions and Variables:

A .....	4-23
B .....	4-28
C .....	4-34
D .....	4-64
E .....	4-84
F .....	4-94

STATEMENTS

G .....	4-106
H .....	4-115
I .....	4-116
K .....	4-131
L .....	4-137
M .....	4-165
N .....	4-173
O .....	4-175
P .....	4-203
R .....	4-236
S .....	4-253
T .....	4-279
U .....	4-284
V .....	4-285
W .....	4-290

# How to Use This Chapter

Descriptions of all the BASIC commands, statements, functions, and variables are included in this chapter. BASIC's built-in functions and variables may be used in any program without further definition.

The first several pages contain lists of all the commands, statements, functions, and variables. These lists may be useful as a quick reference. The rest of the chapter, arranged alphabetically, describes each command, statement, function, and variable in more detail.

The distinction between a command and a statement is largely a matter of tradition. Commands, because they generally operate on programs, are usually entered in direct mode. Statements generally direct program flow from within a program, and so are usually entered in indirect mode as part of a program line. Actually, most BASIC commands and statements can be entered in either direct or indirect mode.

The description of each command, statement, function, or variable in this chapter is formatted as follows:

**Purpose:** Tells what the command, statement, function, or variable does.

**Versions:** Indicates which versions of BASIC allow the command, statement, function, or variable. For example, if you look under "CHAIN Statement" in this chapter, you will note that after **Versions:** it says:

Cassette	Disk	Advanced	Compiler
***	***	(**)	

The asterisks indicate which versions of BASIC support the statement. This example shows that you

can use the CHAIN statement for programs written in the Disk and Advanced versions of BASIC.

In this example you will notice that the asterisks under the word "Compiler" are in parentheses. This means that there are differences between the way the statement works under the BASIC interpreter and the way it works under the IBM Personal Computer BASIC Compiler. The IBM Personal Computer BASIC Compiler is an optional software package available from IBM. If you have the BASIC Compiler, the *IBM Personal Computer BASIC Compiler* manual explains these differences.

**Format:** Shows the correct format for the command, statement, function, or variable. A complete explanation of the syntax format is presented in the Preface. Remember to keep these rules in mind.

- Words in capital letters are keywords and must be entered as shown. They may be entered in any combination of uppercase and lowercase letters. BASIC always converts words to uppercase (unless they are part of a quoted string, remark, or DATA statement).
- You are to supply any items in lowercase italic letters.
- Items in square brackets ([ ]) are optional.
- An ellipsis (...) indicates an item may be repeated as many times as you wish.
- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

**Remarks:** Describes in detail how the command, statement, function, or variable is used.

**Example:** Shows direct mode statements, sample programs, or program segments that demonstrate the use of the command, statement, function, or variable.

In the formats given in this chapter, some of the parameters have been abbreviated as follows:

$x, y, z$  represent any numeric expressions

$i, j, k, m, n$  represent integer expressions

$x\$, y\$$  represent string expressions

$v, w\$$  represent numeric and string variables,  
respectively

If a single- or double-precision value is supplied where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

**Functions and Variables:** In the format description, most of the functions and variables are shown on the right side of an assignment statement. This is to remind you that they are not used like statements and commands. It is not meant to suggest that you are limited to using them in assignment statements. You can use them anywhere you would use a regular variable, *except* on the left side of an assignment statement. Any exceptions are noted in the particular section describing the function or variable. A few of the functions are limited to being used in PRINT statements; these are shown as part of a PRINT statement.

**Note:** Only integer and single-precision results are returned by the numeric functions, except where indicated otherwise.

# Commands

The following is a list of all the commands used in BASIC. The syntax of each command is shown, but not always in its entirety. You can find detailed information about each command in the alphabetical part of this chapter. You may also want to check the next section in this chapter, "Statements," for a list of the BASIC statements.

Command	Action
<b>AUTO number, increment</b>	Generates line numbers automatically.
<b>BLOAD filespec,offset</b>	Loads binary data (such as a machine language program) into memory.
<b>BSAVE filespec,offset,length</b>	Saves binary data.
<b>CLEAR ,n,m</b>	Clears program variables, and optionally sets memory area.
<b>CONT</b>	Continues program execution.
<b>DELETE line1-line2</b>	Deletes specified program lines.
<b>EDIT line</b>	Displays a program line for changing.
<b>FILES filespec</b>	Lists files in the diskette directory that match a file specification.
<b>KILL filespec</b>	Erases a diskette file.

Command	Action
<b>LIST line1-line2,filespec</b>	Lists program lines on the screen or to the specified file.
<b>LLIST line1-line2</b>	Lists program lines on the printer.
<b>LOAD filespec</b>	Loads a program file. Can include the R option to run it.
<b>MERGE filespec</b>	Merges a saved program with the program in memory.
<b>NAME filespec AS filename</b>	Renames a diskette file.
<b>NEW</b>	Erases the current program and variables.
<b>RENUM newnum,oldnum,increment</b>	Renumbers program lines.
<b>RESET</b>	Reinitializes diskette information. Similar to CLOSE.
<b>RUN filespec</b>	Executes a program. The R option may be used to keep files open.
<b>RUN line</b>	Runs the program in memory starting at the specified line.
<b>SAVE filespec</b>	Saves the program in memory under the given filename. A or P option saves in ASCII or protected format.
<b>SYSTEM</b>	Ends BASIC. Closes all files and returns to DOS.
<b>TRON, TROFF</b>	Turns trace on or off.

# Statements

This section lists all the BASIC statements alphabetically in two categories: I/O (Input/Output) Statements and Non-I/O Statements. The list tells what each statement does and shows the syntax. For the more complex statements the syntax shown may not be complete. You can find detailed information about each statement in the alphabetical portion of this chapter, later on.

You may also want to look at the previous section, "Commands," for a list of the BASIC commands.

## Non-I/O Statements

Statement	Action
<b>CALL numvar(variable list)</b>	Calls a machine language program.
<b>CHAIN filespec</b>	Calls a program and passes variables to it. Other options allow you to use overlays, begin running at a line other than the first line, pass all variables, or delete an overlay.
<b>COM(n) ON/OFF/STOP</b>	Enables and disables trapping of communications activity.
<b>COMMON list of variables</b>	Passes variables to a chained program.
<b>DATE\$ = x\$</b>	Sets the date.
<b>DEF FNname(arg list)=expression</b>	Defines a numeric or string function.

Statement	Action
<b>DEF</b> type ranges of letters	Defines default variable types, where <i>type</i> is INT, SNG, DBL, or STR.
<b>DEF SEG=</b> address	Defines current segment of memory.
<b>DEF USRn=</b> offset	Defines starting address for machine language subroutine n.
<b>DIM</b> list of subscripted variables	Declares maximum subscript values for arrays and allocates space for them.
<b>END</b>	Stops the program, closes all files, and returns to command level.
<b>ERASE</b> arraynames	Eliminates arrays from a program.
<b>ERROR</b> n	Simulates error number n.
<b>FOR</b> variable= <b>x</b> TO y <b>STEP</b> z	Repeats program lines a number of times. The NEXT statement closes the loop.
<b>GOSUB</b> line	Calls a subroutine by branching to the specified line. The RETURN statement returns from the subroutine.
<b>GOTO</b> line	Branches to the specified line.

Statement	Action
<b>IF expression THEN clause ELSE clause</b>	Performs the statement(s) in the THEN clause if expression is true (nonzero). Otherwise, performs the ELSE clause or goes to the next line.
<b>KEY ON/OFF/LIST</b>	Displays soft keys or turns display off.
<b>KEY n, x\$</b>	Sets soft key n to the value of the string x\$.
<b>KEY(n) ON/OFF/STOP</b>	Enables/disables trapping of function keys or cursor control keys.
<b>LET variable=expression</b>	Assigns the value of the expression to the variable.
<b>MID\$(v\$,n,m)=y\$</b>	Replaces part of the variable v\$ with the string y\$, starting at position n and replacing m characters.
<b>MOTOR state</b>	Turns cassette motor on if state is nonzero, off if state is zero.
<b>NEXT variable</b>	Closes a FOR...NEXT loop (see FOR).
<b>ON COM(n) GOSUB line</b>	Enables trap routine for communications activity.
<b>ON ERROR GOTO line</b>	Enables error trap routine beginning at line specified.

Statement	Action
ON n GOSUB line list	Branches to subroutine specified by n.
ON n GOTO line list	Branches to statement specified by n.
ON KEY(n) GOSUB line	Enables trap routine for the specified function key or cursor control key.
ON PEN GOSUB line	Enables trap routine for light pen.
ON STRIG(n) GOSUB line	Enables trap routine for joystick button.
OPTION BASE n	Specifies the minimum value for array subscripts.
PEN ON/OFF/STOP	Enables/disables the light pen function.
POKE n,m	Puts byte m into memory at the location specified by n.
RANDOMIZE n	Reseeds the random number generator.
REM remark	Includes remark in program.
RESTORE line	Resets DATA pointer so DATA statements may be reread.
RESUME line/NEXT/0	Returns from error trap routine.

Statement	Action
<b>RETURN</b> line	Returns from subroutine.
<b>STOP</b>	Stops program execution, prints a break message, and returns to command level.
<b>STRIG ON/OFF</b>	Enables/disables joystick button function.
<b>STRIG(n) ON/OFF/STOP</b>	Enables/disables joystick button trapping.
<b>SWAP variable1,variable2</b>	Exchanges values of two variables.
<b>TIME\$ = x\$</b>	Sets the time.
<b>WAIT port,n,m</b>	Suspends program execution until the specified port develops the specified bit pattern.
<b>WEND</b>	Closes a WHILE...WEND loop (see WHILE).
<b>WHILE expression</b>	Begins a loop which executes as long as the expression is true.

# I/O Statements

Statement	Action
BEEP	Beeps the speaker.
CIRCLE (x,y),r	Draws a circle with center (x,y) and radius r. Other options allow you to specify a part of the circle to be drawn, or to change the aspect ratio to draw an ellipse.
CLOSE #f	Closes a file.
CLS	Clears the screen.
COLOR foreground,background,border	In text mode, sets colors for foreground, background, and the border screen.
COLOR background,palette	In graphics mode, sets background color and palette of foreground colors.
DATA list of constants	Creates a data table to be used by READ statements.
DRAW string	Draws a figure as specified by string.
FIELD #f,width AS stringvar...	Defines fields in a random file buffer.
GET #f,number	Reads a record from a random file.
GET (x1,y1)-(x2,y2),arrayname	Reads graphic information from screen.

Statement	Action
<b>INPUT "prompt";variable list</b>	Reads data from the keyboard.
<b>INPUT #f,variable list</b>	Reads data from file f.
<b>LINE (x1,y1)-(x2,y2)</b>	Draws a line on the screen. Other parameters allow you to draw a box, and fill the box in.
<b>LINE INPUT "prompt";stringvar</b>	Reads an entire line from the keyboard, ignoring commas or other delimiters.
<b>LINE INPUT #f,stringvar</b>	Reads an entire line from a file.
<b>LOCATE row,col</b>	Positions the cursor. Other parameters allow you to define the size of the cursor and whether it is visible or not.
<b>LPRINT list of expressions</b>	Prints data on the printer.
<b>LPRINT USING v\$;list of expressions</b>	Prints data on the printer using the format specified by v\$.
<b>LSET stringvar=x\$</b>	Left-justifies a string in a field.
<b>OPEN filespec FOR mode AS #f</b>	Opens the file for the mode specified. Another option sets the record length for random files.

Statement	Action
<b>OPEN mode,#f,filespec,recl</b>	Alternative form of preceding OPEN.
<b>OPEN "COMn:options" AS #f</b>	Opens file for communications.
<b>OUT n,m</b>	Outputs the byte m to the machine port n.
<b>PAINT (x,y),paint,boundary</b>	Fills in an area on the screen defined by boundary with the paint color.
<b>PLAY string</b>	Plays music as specified by string.
<b>PRINT list of expressions</b>	Displays data on the screen.
<b>PRINT USING v\$,list of expressions</b>	Displays data using the format specified by v\$.
<b>PRINT #f, list of exps</b>	Writes the list of expressions to file f.
<b>PRINT #f, USING v\$;list of exps</b>	Writes data to file f using the format specified by v\$.
<b>PRESET (x,y)</b>	Draws a point on the screen in background color. See PSET.
<b>PSET (x,y),color</b>	Draws a point on the screen, in the foreground color if color is not specified.
<b>PUT #f,number</b>	Writes data from a random file buffer to the file.

<b>Statement</b>	<b>Action</b>
<b>PUT (x,y),array,action</b>	Writes graphic information to the screen.
<b>READ variable list</b>	Retrieves information from the data table created by DATA statements.
<b>RSET stringvar=x\$</b>	Right-justifies a string in a field. See LSET.
<b>SCREEN mode,burst,apage,vpage</b>	Sets screen mode, color on or off, display page, and active page.
<b>SOUND freq,duration</b>	Generates sound through the speaker.
<b>WIDTH size</b>	Sets screen width. Other options allow you to specify the width of a printer or a communications file.
<b>WRITE list of expressions</b>	Outputs data on the screen.
<b>WRITE #f, list of expressions</b>	Outputs data to a file.

# Functions and Variables

The built-in functions and variables available in BASIC are listed below, grouped into two general categories: numeric functions, or those which return a numeric result; and string functions, or those which return a string result.

Each category is further subdivided according to the usage of the functions. The numeric functions are divided into general arithmetic (or algebraic) functions; string-related functions, which operate on strings; and input/output and miscellaneous functions. The string functions are separated into general string functions, and input/output and miscellaneous string functions.

**Note:** Only integer and single-precision results are returned by the numeric functions, except where indicated otherwise.

## Numeric Functions (return a numeric value)

### ARITHMETIC

Function	Result
<b>ABS(x)</b>	Returns the absolute value of x.
<b>ATN(x)</b>	Returns the arctangent (in radians) of x.
<b>CDBL(x)</b>	Converts x to a double-precision number.
<b>CINT(x)</b>	Converts x to an integer by rounding.
<b>COS(x)</b>	Returns the cosine of angle x, where x is in radians.
<b>CSNG(x)</b>	Converts x to a single-precision number.

Function	Result
<b>EXP(x)</b>	Raises $e$ to the $x$ power.
<b>FIX(x)</b>	Truncates $x$ to an integer.
<b>INT(x)</b>	Returns the largest integer less than or equal to $x$ .
<b>LOG(x)</b>	Returns the natural logarithm of $x$ .
<b>RND(x)</b>	Returns a random number.
<b>SGN(x)</b>	Returns the sign of $x$ .
<b>SIN(x)</b>	Returns the sine of angle $x$ , where $x$ is in radians.
<b>SQR(x)</b>	Returns the square root of $x$ .
<b>TAN(x)</b>	Returns the tangent of angle $x$ , where $x$ is in radians.

For information on how to calculate mathematical functions which are not included in this list, refer to "Appendix E. Mathematical Functions."

## STRING-RELATED

Function	Result
<b>ASC(x\$)</b>	Returns the ASCII code for the first character in $x$$ .
<b>CVI(x\$), CVS(x\$), CVD(x\$)</b>	Converts $x$$ to a number of the indicated precision.
<b>INSTR(n,x\$,y\$)</b>	Returns the position of first occurrence of $y$$ in $x$$ starting at position $n$ .
<b>LEN(x\$)</b>	Returns the length of $x$$ .
<b>VAL(x\$)</b>	Returns the numeric value of $x$$ .

## I/O and MISCELLANEOUS

Function	Result
CSRLIN	Returns the vertical line position of the cursor.
EOF(f)	Indicates an end of file condition on file f.
ERL	Returns the line number where the last error occurred (see ERR).
ERR	Returns the error code number of the last error.
FRE(x\$)	Returns the amount of free space in memory not currently in use by BASIC.
INP(n)	Reads a byte from port n.
LOC(f)	Returns the "location" of file f: <ul style="list-style-type: none"><li>● next record number of random file</li><li>● number of sectors read or written for sequential file</li><li>● number of characters in communications input buffer</li></ul>
LOF(f)	Returns the length of file f: <ul style="list-style-type: none"><li>● number of bytes (in multiples of 128) in sequential or random file</li><li>● number of bytes free in communications input buffer</li></ul>
LPOS(n)	Returns the carriage position of the printer.

Function	Result
<b>PEEK(n)</b>	Reads the byte in memory location n.
<b>PEN(n)</b>	Reads the light pen.
<b>POINT(x,y)</b>	Returns the color of point (x,y) (graphics mode).
<b>POS(n)</b>	Returns the cursor column position.
<b>SCREEN(row,col,z)</b>	Returns the character or color at position (row,col).
<b>STICK(n)</b>	Returns the coordinates of a joystick.
<b>STRIG(n)</b>	Returns the state of a joystick button.
<b>USRn(x)</b>	Calls a machine language subroutine with argument x.
<b>VARPTR(variable)</b>	Returns the address of the variable in memory.
<b>VARPTR(#f)</b>	Returns the address of the file control block for file f.

# String Functions (return a string value)

## GENERAL

Function	Result
<b>CHR\$(n)</b>	Returns the character with ASCII code n.
<b>LEFT\$(x\$,n)</b>	Returns the leftmost n characters of x\$.
<b>MID\$(x\$,n,m)</b>	Returns m characters from x\$ starting at position n.
<b>RIGHT\$(x\$,n)</b>	Returns the rightmost n characters of x\$.
<b>SPACE\$(n)</b>	Returns a string of n spaces.
<b>STRING\$(n,m)</b>	Returns the character with ASCII value m, repeated n times.
<b>STRING\$(n,x\$)</b>	Returns the first character of x\$ repeated n times.

## I/O and MISCELLANEOUS

Function	Result
<b>DATE\$</b>	Returns the system date.
<b>HEX\$(n)</b>	Converts n to a hexadecimal string.
<b>INKEY\$</b>	Reads a character from the keyboard.
<b>INPUT\$(n,#f)</b>	Reads n characters from file f.

STATEMENTS

Function	Result
<b>MKI\$(x), MKS\$(x), MKD\$(x)</b>	Converts x in indicated precision to proper length string.
<b>OCT\$(n)</b>	Converts n to an octal string.
<b>SPC(n)</b>	Prints n spaces in a PRINT or LPRINT statement.
<b>STR\$(x)</b>	Converts x to a string value.
<b>TAB(n)</b>	Tabs to position n in a PRINT or LPRINT statement.
<b>TIME\$</b>	Returns the system time.
<b>VARPTR\$(v)</b>	Returns a three-byte string containing the type of variable, and the address of the variable in memory.

# ABS Function

---

**Purpose:** Returns the absolute value of the expression  $x$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{ABS}(x)$

**Remarks:**  $x$  may be any numeric expression.

The absolute value of a number is always positive or zero.

**Example:** 0k  
PRINT ABS(7\*(-5))  
35  
0k

The absolute value of -35 is positive 35.

STATEMENTS

# ASC Function

---

**Purpose:** Returns the ASCII code for the first character of the string  $x\$$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{ASC}(x\$)$

**Remarks:**  $x\$$  may be any string expression.

The result of the ASC function is a numerical value that is the ASCII code of the first character of the string  $x\$$ . (See “Appendix G. ASCII Character Codes” for ASCII codes.) If  $x\$$  is null, an “Illegal function call” error is returned.

The CHR\$ function is the inverse of the ASC function, and it converts the ASCII code to a character.

**Example:**

```
0k
1Ø X$ = "TEST"
2Ø PRINT ASC(X$)
RUN
 84
0k
```

This example shows that the ASCII code for a capital T is 84. Print ASC(“TEST”) would work just as well.

# ATN Function

---

**Purpose:** Returns the arctangent of  $x$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{ATN}(x)$

**Remarks:**  $x$  may be a numeric expression of any numeric type, but the evaluation of ATN is always performed in single precision.

The ATN function returns the angle whose tangent is  $x$ . The result is a value in radians in the range -PI/2 to PI/2, where PI=3.141593.

If you want to convert radians to degrees, multiply by 180/PI.

STATEMENTS

**Example:** 0k

```
PRINT ATN(3)  
1.249046
```

0k

```
10 PI=3.141593  
20 RADIANS=ATN(1)  
30 DEGREES=RADIANS*180/PI  
40 PRINT RADIANS,DEGREES  
RUN  
.7853983          45  
0k
```

The first example shows the use of the ATN function to calculate the arctangent of 3. The second example finds the angle whose tangent is 1. It is .7853983 radians, or 45 degrees.

# AUTO Command

---

**Purpose:** Generates a line number automatically each time you press Enter.

**Versions:** Cassette      Disk      Advanced      Compiler

\*\*\*      \*\*\*      \*\*\*

**Format:** AUTO [*number*] [, [*increment*] ]

**Remarks:** *number* is the number which will be used to start numbering lines. A period (.) may be used in place of the line number to indicate the current line.

*increment* is the value that will be added to each line number to get the next line number.

Numbering begins at *number* and increments each subsequent line number by *increment*. If both values are omitted, the default is 10,10. If *number* is followed by a comma but *increment* is not specified, the last increment specified in an AUTO command is assumed. If *number* is omitted but *increment* is included, then line numbering begins with 0.

AUTO is usually used for entering programs. It releases you from having to type each line number.

# AUTO Command

If AUTO generates a line number that already exists in the program, an asterisk (\*) is printed after the number to warn you that any input will replace the existing line. However, if you press Enter immediately after the asterisk, the existing line will not be replaced and AUTO will generate the next line number.

AUTO ends when you press Ctrl-Break. The line in which Ctrl-Break is typed is not saved. After a Ctrl-Break, BASIC returns to command level.

**Note:** When in AUTO mode, you may make changes only to the current line. If you want to change another line on the screen, be sure to exit AUTO by first pressing Ctrl-Break.

**Example:** AUTO

This command generates line numbers 10, 20, 30, 40, ...

AUTO 100,50

This generates line numbers 100, 150, 200, ...

AUTO 500,

This generates line numbers 500, 550, 600, 650, ...  
The increment is 50 since 50 was the increment in the previous AUTO command.

AUTO ,20

This generates line numbers 0, 20, 40, 60, ...

# BEEP Statement

---

**Purpose:** Beeps the speaker.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** BEEP

**Remarks:** The BEEP statement sounds the speaker at 800 Hz for 1/4 second. BEEP has the same effect as:

PRINT CHR\$(7);

**Example:** 2430 IF X < 20 THEN BEEP

In this example, the program checks to see if X is out of range. If it is, the computer "complains" by beeping.

# BLOAD Command

---

**Purpose:** Loads a memory image file into memory.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** BLOAD *filespec* [,*offset*]

**Remarks:** *filespec* is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3, otherwise a "Bad file name" error occurs and the load is cancelled.

*offset* is a numeric expression in the range 0 to 65535. This is the address at which loading is to start, specified as an offset into the segment declared by the last DEF SEG statement.

If *offset* is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location it was saved from.

When a BLOAD command is executed, the named file is loaded into memory starting at the specified location. If the file is to be loaded from the device CAS1:, the cassette motor is turned on automatically.

If you are using Cassette BASIC and the device named is omitted, CAS1: is assumed. CAS1: is the only allowable device for BLOAD in Cassette BASIC. If you are using Disk or Advanced BASIC and the device name is omitted, the DOS default diskette drive is used.

# BLOAD Command

BLOAD and BSAVE are useful for loading and saving machine language programs. (You may perform machine language programs from within a BASIC program by using the CALL statement.) However, BLOAD and BSAVE are not restricted to machine language programs. Any segment may be specified as the target or source for these statements via the DEF SEG statement. You have a useful way of saving and displaying screen images: save from or load to the screen buffer.

## Warning:

BASIC does not do any checking on the address. That is, it is possible to BLOAD anywhere in memory. You should not BLOAD over BASIC's stack, BASIC's variable area, or your BASIC program.

## Notes when using CAS1:

1. If you enter the BLOAD command in direct mode, the file names on the tape will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message "Skipped." for the files not matching the named file, and "Found." when the named file is found. Types of files and the associated letter are:
  - .B for BASIC programs in internal format (created with SAVE command)
  - .P for protected BASIC programs in internal format (created with SAVE ,P command)
  - .A for BASIC programs in ASCII format (created with SAVE ,A command)
  - .M for memory image files (created with BSAVE command)
  - .D for data files (created by OPEN followed by output statements)

# BLOAD Command

If the BLOAD command is executed in a BASIC program, the file names skipped and found are not displayed on the screen.

2. You may press Ctrl-Break any time during BLOAD. This will cause BASIC to exit the search and return to direct mode between files or after a time-out period. Previous memory contents do not change.
3. If CAS1: is specified as the device and the filename is omitted, the next memory image (.M) file on the tape is loaded.

**Example:**

```
10 'load the screen buffer
20 'point SEG at screen buffer
30 DEF SEG= &HB800
40 'load PICTURE into screen buffer
50 BLOAD "PICTURE",0
```

This example loads the screen buffer for the Color/Graphics Monitor Adapter, which is at absolute address hex B8000. If you were loading the screen buffer for the IBM Monochrome Display and Parallel Printer Adapter, you would have to change line 30 to read &HB000 (the actual address is hex B0000). Note that the DEF SEG statement in 30 and the offset of 0 in 50 is wise. This assures that the correct address is used.

The example for BSAVE in the next section illustrates how PICTURE was saved.

# BSAVE Command

---

**Purpose:** Saves portions of the computer's memory on the specified device.

**Versions:** Cassette      Disk      Advanced      Compiler

\*\*\*

\*\*\*

\*\*\*

\*\*\*

**Format:** BSAVE *filespec,offset,length*

**Remarks:** *filespec* is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3; otherwise, a "Bad file name" error occurs and the save is cancelled.

*offset* is a numeric expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG. Saving will start from this position.

*length* is a numeric expression in the range 1 to 65535. This is the length of the memory image to be saved.

If *offset* or *length* is omitted, a "Syntax error" will occur and the save will be cancelled.

If the device name is omitted in Cassette BASIC, CAS1: is assumed. CAS1: is the only allowable device for BSAVE in Cassette BASIC. In Disk and Advanced BASIC, if the device name is omitted, the DOS default diskette drive is used.

If you are saving the CAS1:, the cassette motor will be turned on and the memory image file will be immediately written to the tape.

# BSAVE Command

BLOAD and BSAVE are useful for loading and saving machine language programs (which can be called using the CALL statement). However, BLOAD and BSAVE are not restricted to machine language programs. By using the DEF SEG statement, any segment may be specified as the target or source for these statements. For example, you can save an image of the screen by doing a BSAVE of the screen buffer.

**Example:**

```
10 'Save the color screen buffer
15 'point segment at screen buffer
20 DEF SEG= &HB800
25 'save buffer in file PICTURE
30 BSAVE "PICTURE",0,&H4000
```

As explained under “BLOAD Command” in the previous section, the address of the 16K screen buffer for the Color/Graphics Monitor Adapter is hex B8000. The address of the 4K screen buffer for the IBM Monochrome Display and Parallel Printer Adapter is hex B0000.

The DEF SEG statement must be used to set up the segment address to the start of the screen buffer. Offset of 0 and length &H4000 specifies that the entire 16K screen buffer is to be saved.

# CALL Statement

---

**Purpose:** Calls a machine language subroutine.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            (\*\*\*)

**Format:** CALL *numvar* [ (*variable* [,*variable*]...)]

**Remarks:** *numvar* is the name of a numeric variable. The value of the variable indicates the starting memory address of the subroutine being called as an offset into the current segment of memory (as defined by the last DEF SEG statement).

*variable* is the name of a variable which is to be passed as an argument to the machine language subroutine.

The CALL statement is one way of interfacing machine language programs with BASIC. The other way is by using the USR function. Refer to "Appendix C. Machine Language Subroutines" for specific considerations about using machine language subroutines.

**Example:** 100 DEF SEG=&H8000  
110 OZ=0  
120 CALL OZ(A,B\$,C)

Line 100 sets the segment to location hex 80000. OZ is set to zero so that the call to OZ will execute the subroutine at location hex 80000. The variables A, B\$, and C are passed as arguments to the machine language subroutine.

# CDBL Function

---

**Purpose:** Converts  $x$  to a double-precision number.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{CDBL}(x)$

**Remarks:**  $x$  may be any numeric expression.

Rules for converting from one numeric precision to another are followed as explained in "How BASIC Converts Numbers from One Precision to Another" in Chapter 3. Refer also to the CINT and CSNG functions for converting numbers to integer and single-precision.

**Example:**

```
0k
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
454.67 454.6699829101563
0k
```

STATEMENTS

The value of CDBL(A) is only accurate to the second decimal place after rounding. The extra digits have no meaning. This is because only two decimal places of accuracy were supplied with A.

# CHAIN Statement

---

**Purpose:** Transfers control to another program, and passes variables to it from the current program.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*                \*\*\*                (\*\*)

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

**Remarks:** *filespec* follows the rules for file specifications outlined in "Naming Files" in Chapter 3. The filename is the name of the program that is transferred to. Example:

```
CHAIN "A:PROG1"
```

*line* is a line number or an expression that evaluates to a line number in the chained-to program. It specifies the line at which the chained-to program is to begin running. If it is omitted, execution begins at the first line in the chained-to program. Example:

```
CHAIN "A:PROG1", 1000
```

*line* (1000 in this example) is not affected by a RENUM command. If PROG1 is renumbered, this example CHAIN statement should be changed to point to the new line number.

ALL specifies that every variable in the current program is to be passed to the chained-to program. If the ALL option is omitted, you must include a COMMON statement in the chaining program to pass variables to the chained-to program. See "COMMON Statement" in this chapter. Example:

```
CHAIN "A:PROG1", 1000, ALL
```

# CHAIN Statement

MERGE brings a section of code into the BASIC program as an overlay. That is, a MERGE operation is performed with the chaining program and the chained-to program. The chained-to program must be an ASCII file if it is to be merged. Example:

```
CHAIN MERGE "A:OVRLAY",1000
```

After using an overlay, you will usually want to delete it so that a new overlay may be brought in. To do this, use the DELETE option, which behaves like the DELETE command. As in the DELETE command, the line numbers specified as the first and last line of the range must exist, or an “Illegal function call” error occurs. Example:

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

This example will delete lines 1000 through 5000 of the chaining program before loading in the overlay (chained-to program). The line numbers in *range* are affected by the RENUM command.

## Notes:

1. The CHAIN statement leaves files open.
2. The CHAIN statement with MERGE option preserves the current OPTION BASE setting.
3. If the MERGE option is omitted, the OPTION BASE setting is not preserved in the chained-to program. Also, without MERGE, CHAIN does not preserve variable types or user-defined functions for use by the chained-to program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

# CHR\$ Function

---

**Purpose:** Converts an ASCII code to its character equivalent.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** *v\$* = CHR\$(*n*)

**Remarks:** *n* must be in the range 0 to 255.

The CHR\$ function returns the one-character string with ASCII code *n*. (ASCII codes are listed in “Appendix G. ASCII Character Codes.”) CHR\$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR\$(7) as a preface to an error message (instead of using BEEP). Look under “ASC Function,” earlier in this chapter, to see how to convert a character back to its ASCII code.

**Example:** 0k  
PRINT CHR\$(66)  
B  
0k

The next example sets function key F1 to the string “AUTO” joined with Enter. This is a good way to set the function keys so the Enter is automatically done for you when you press the function key.

0k  
KEY 1, "AUTO"+CHR\$(13)  
0k

# CHR\$ Function

The following example is a program which shows all the displayable characters, along with their ASCII codes, on the screen in 80-column width. It can be used with either the IBM Monochrome Display and Parallel Printer Adapter or the Color/Graphics Monitor Adapter.

```
10 CLS
20 FOR I=1 TO 255
30 ' ignore nondisplayable characters
40 IF (I>6 AND I<14) OR (I>27 AND I<32) THEN 100
50 COLOR 0,7 ' black on white
60 PRINT USING "###"; I ; ' 3-digit ASCII code
70 COLOR 7,0 ' white on black
80 PRINT " "; CHR$(I); " ";
90 IF POS(0)>75 THEN PRINT ' go to next line
100 NEXT I
```

STATEMENTS

# CINT Function

---

**Purpose:** Converts  $x$  to an integer.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{CINT}(x)$

**Remarks:**  $x$  may be any numeric expression. If  $x$  is not in the range -32768 to 32767, an “Overflow” error occurs.

$x$  is converted to an integer by rounding the fractional portion.

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

**Example:** 0k  
PRINT CINT(45.67)  
46  
0k  
PRINT CINT(-2.89)  
-3  
0k

Observe in both examples how rounding occurs.

# CIRCLE Statement

---

**Purpose:** To draw an ellipse on the screen with center  $(x,y)$  and radius  $r$ .

**Versions:** Cassette    Disk    Advanced    Compiler  
                      \*\*\*                                    \*\*\*

Graphics mode only.

**Format:** CIRCLE  $(x,y),r$  [,color [,start,end [,aspect] ]]

**Remarks:**  $(x,y)$  are the coordinates of the center of the ellipse. The coordinates may be given in either absolute or relative form. See “Specifying Coordinates” under “Graphics Modes” in Chapter 3.

$r$  is the radius (major axis) of the ellipse in points.

*color* is a number which specifies the color of the ellipse, in the range 0 to 3. In medium resolution, *color* selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution, a *color* of 0 (zero) indicates black, and the default of 1 (one) indicates white.

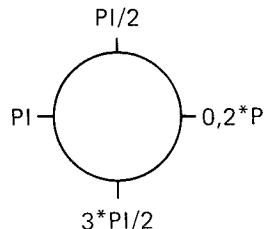
*start*,  
*end* are angles in radians and may range from -2\*PI to 2\*PI, where PI=3.141593.

*aspect* is a numeric expression.

# CIRCLE

## Statement

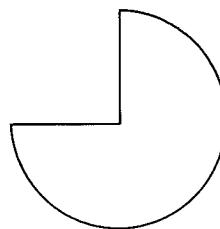
*start* and *end* specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:



If the start or end angle is negative (-0 is not allowed), the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (note that this is not the same as adding  $2\pi$ ). The start angle may be greater or less than the end angle. For example,

```
1Ø PI=3.141593  
2Ø SCREEN 1  
3Ø CIRCLE (16Ø,10Ø),6Ø,, -PI,-PI/2
```

will draw a part of a circle similar to the following:



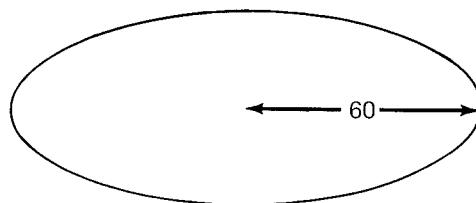
*aspect* affects the ratio of the x-radius to the y-radius. The default for *aspect* is 5/6 in medium resolution and 5/12 in high resolution. These values give a visual circle assuming the standard screen aspect ratio of 4/3.

# CIRCLE Statement

If *aspect* is less than one, then *r* is the x-radius. That is, the radius is measured in points in the horizontal direction. If *aspect* is greater than one, then *r* is the y-radius. For example,

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

will draw an ellipse like this:



In many cases, an *aspect* of 1 (one) will give nicer looking circles in medium resolution. This will also cause the circle to be drawn somewhat faster.

The last point referenced after a circle is drawn is the center of the circle.

Points that are off the screen are not drawn by CIRCLE.

**Example:** The following example draws a face.

```
10 PI=3.141593  
20 SCREEN 1 ' medium res. graphics  
30 COLOR 0,1 ' black background, palette 1  
40 'two circles in color 1 (cyan)  
50 CIRCLE (120,50),10,1  
60 CIRCLE (200,50),10,1  
70 'two horizontal ellipses  
80 CIRCLE (120,50),30,,,5/18  
90 CIRCLE (200,50),30,,,5/18  
100 'arc in color 2 (magenta)  
110 CIRCLE (160,0),150,2, 1.3*PI, 1.7*PI  
120 'arc, one side connected to center  
130 CIRCLE (160,52),50,, 1.4*PI, -1.6*PI
```

# CLEAR Command

---

**Purpose:** Sets all numeric variables to zero and all string variables to null. Options set the end of memory and the amount of stack space.

**Versions:** Cassette      Disk      Advanced      Compiler  
              \*\*\*          \*\*\*          \*\*\*          (\*\*\*)

**Format:** CLEAR [,*n*] [,*m*]

**Remarks:** *n* is a byte count which, if specified, sets the maximum number of bytes for the BASIC workspace (where your program and data are stored, along with the interpreter workarea). You would probably include *n* if you need to reserve space in storage for machine language programs.

*m* sets aside stack space for BASIC. The default is 512 bytes, or one-eighth of the available memory (whichever is smaller). You may want to include *m* if you use a lot of nested GOSUB statements or FOR...NEXT loops in your program, or if you use PAINT to do complex scenes.

CLEAR frees all memory used for data without erasing the program which is currently in memory. After a CLEAR, arrays are undefined; numeric variables have a value of zero; string variables have a null value; and any information set with any DEF statement is lost. (This includes DEF FN, DEF SEG, and DEF USR, as well as DEFINT, DEFDBL, DEFSNG, and DEFSTR.)

# CLEAR Command

Executing a CLEAR command turns off any sound that is running and resets to Music Foreground. Also, PEN and STRIG are reset to OFF.

The ERASE statement may be useful to free some memory without erasing all the data in the program. It erases only specified arrays from the work area. Refer to "ERASE Statement" in this chapter for details.

**Example:** This example clears all data from memory (without erasing the program):

```
CLEAR
```

The next example clears the data and sets the maximum workspace size to 32K-bytes:

```
CLEAR ,32768
```

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

```
CLEAR , ,2000
```

The last example clears data, sets the maximum workspace for BASIC to 32K-bytes, and sets the stack size to 2000 bytes:

```
CLEAR ,32768,2000
```

# CLOSE Statement

---

**Purpose:** Concludes I/O to a device or file.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

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

**Remarks:** *filenum* is the number used on the OPEN statement.

The association between a particular file or device and its file number stops when CLOSE is executed. Subsequent I/O operations specifying that file number will be invalid. The file or device may be opened again using the same or a different file number; or the file number may be reused to open any device or file.

A CLOSE to a file or device opened for sequential output causes the final buffer to be written to the file or device.

A CLOSE with no file numbers specified causes all devices and files that have been opened to be closed.

Executing an END, NEW, RESET, SYSTEM or RUN without the R option causes all open files and devices to be automatically closed. STOP does not close any files or devices.

Refer also to “OPEN Statement” in this chapter for information about opening files.

# CLOSE Statement

**Example:** 100 CLOSE 1, #2, #3

Causes the files and devices associated with file numbers 1, 2, and 3 to be closed.

200 CLOSE

Causes all open devices and files to be closed.

# **CLS Statement**

---

**Purpose:** Clears the screen.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** CLS

**Remarks:** If the screen is in text mode, the active page (see "SCREEN Statement" in this chapter) is cleared to the background color (see "COLOR Statement," also in this chapter).

If the screen is in graphics mode (medium or high resolution), the entire screen buffer is cleared to the background color.

The CLS statement also returns the cursor to the home position. In text mode, this means the cursor is located in the upper left-hand corner of the screen. In graphics mode, this means the "last referenced point" for future graphics statements is the point in the center of the screen ((160,100) in medium resolution, (320,100) in high resolution).

Changing the screen mode or width by using the SCREEN or WIDTH statements also clears the screen. The screen may also be cleared by pressing Ctrl-Home.

**Example:** 10 COLOR 10,1  
20 CLS

With the Color/Graphics Monitor Adapter, this example clears the screen to Blue.

# COLOR Statement

---

**Purpose:** Sets the colors for the foreground, background, and border screen. Refer to "Text Mode" in Chapter 3 for an explanation of these terms.

The syntax of the COLOR statement depends on whether you are in text mode or graphics mode, as set by the SCREEN statement.

In text mode, you can set the following:

Foreground- 1 of 16 colors  
Character blink, if desired  
Background- 1 of 8 colors  
Border- 1 of 16 colors

You can set the following in medium resolution graphics mode:

Background- 1 of 16 colors  
Palette- 1 of 2 palettes with 3 colors each  
The border is the same as the background color.

STATEMENTS

## The COLOR Statement in Text Mode

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

Text mode only.

**Format:** COLOR [*foreground*] [, [*background*] [, *border*]]

# COLOR Statement (Text)

**Remarks:** *foreground* is a numeric expression in the range 0 to 31, representing the character color.

*background* is a numeric expression in the range 0 to 7 for the background color.

*border* is a numeric expression in the range 0 to 15. It is the color for the border screen.

If you have the Color/Graphics Monitor Adapter, the following colors are allowed for *foreground*:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

Colors and intensity may vary depending on your display device.

You might like to think of colors 8 to 15 as "light" or "high-intensity" values of colors 0 to 7.

You can make the characters blink by setting *foreground* equal to 16 plus the number of the desired color. That is, a value of 16 to 31 causes blinking characters.

You may select only colors 0 through 7 for *background*.

# COLOR Statement (Text)

If you have the IBM Monochrome Display and Parallel Printer Adapter, the following values can be used for *foreground*:

- 0 Black
- 1 Underlined character with white foreground
- 2-7 White

In a manner similar to the Color/Graphics Monitor Adapter, adding 8 to the number of the desired color gives you the color in high-intensity. For example, a value of 15 gives you high-intensity white. A value of 9 gives you high-intensity white, underlined. You can't make high-intensity black.

As with the Color/Graphics Monitor Adapter, you can make the character blink by adding 16 to the number of the desired color. Thus, 16 gives you black blinking characters, and 31 gives you high-intensity white blinking characters.

For *background* with the IBM Monochrome Display and Parallel Printer Adapter, you may select the following values:

- 0-6 Black
- 7 White

**Note:** White (color 7) as a background color shows up as white on the IBM Monochrome Display only when it is used with a foreground color of 0, 8, 16, or 24 (black). This creates reverse image characters.

Black (color 0, 8, 16, or 24) as a foreground color shows up as black only when used with a background color of 0 (which makes the characters invisible) or 7 (which creates reverse image characters).

Other combinations of foreground and background colors produce standard (white on black) results on the IBM Monochrome Display.

# COLOR Statement (Text)

Notes for either adapter:

1. Foreground color may equal background color. This has the effect of making any character displayed invisible. Changing the foreground or background color will make subsequent characters visible again.
2. Any parameter may be omitted. Omitted parameters assume the old value.
3. If the COLOR statement ends in a comma (,), you will get a “Missing operand” error, but the color will change. For example,

COLOR ,7,

is invalid.

4. Any values entered outside the range 0 to 255 will result in an “Illegal function call” error. Previous values are retained.

**Example:** 10 COLOR 14,1,Ø

This sets a yellow foreground, a blue background, and a black border screen.

# COLOR Statement (Text)

The following example can be used with either the Color/Graphics Monitor Adapter or the IBM Monochrome Display and Parallel Printer Adapter:

```
10 PRINT "Enter your ";
20 COLOR 15,0 'highlight next word
30 PRINT "password";
40 COLOR 7    'return to default (white on black)
50 PRINT " here: ";
60 COLOR 0    'invisible (black on black)
70 INPUT PASSWORD$
80 IF PASSWORD$="secret" THEN 120
90 ' blink and highlight error message
100 COLOR 31: PRINT "Wrong Password": COLOR 7
110 GOTO 10
120 COLOR 0,7 'reverse image (black on white)
130 PRINT "Program continues...";
140 COLOR 7,0 'return to default (white on black)
```

# COLOR Statement (Graphics)

## The COLOR Statement in Graphics Mode

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

Graphics mode, medium resolution only.

**Format:** COLOR [*background*] [,*palette*]]

**Remarks:** *background* is a numeric expression specifying the background color. The colors allowed for *background* are 0 through 15, as described previously under "The COLOR Statement in Text Mode."

*palette* is a numeric expression which selects the palette of colors.

The colors selected when you choose each palette are as follows:

Color	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

If *palette* is an even number, palette 0 is selected. This associates the colors Green, Red, and Brown to the color numbers 1, 2, and 3. Palette 1 (Cyan/Magenta/White) is selected when *palette* is an odd number.

The color selected for *background* may be the same as any of the palette colors.

# COLOR Statement (Graphics)

Any parameter may be omitted from the COLOR statement. Omitted parameters assume the old value.

In graphics mode, the COLOR statement sets a background color and a palette of three colors. You may select any one of these four colors for display with the PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW statements. It has meaning in medium resolution only (set by SCREEN 1 statement). Using COLOR in high resolution will result in an "Illegal function call" error.

Any values entered outside the range 0 to 255 will result in an "Illegal function call" error. Previous values will be retained.

**Example:** 5 SCREEN 1  
10 COLOR 9,0

Sets the background to light blue, and selects palette 0.

20 COLOR ,1

The background stays light blue, and palette 1 is selected.

# **COM(*n*) Statement**

---

**Purpose:** Enables or disables trapping of communications activity to the specified communications adapter.

**Versions:** Cassette      Disk      Advanced      Compiler  
                                \*\*\*                                (\*\*)

**Format:**    COM(*n*) ON  
                  COM(*n*) OFF  
                  COM(*n*) STOP

**Remarks:** *n*      is the number of the communications adapter  
(1 or 2).

A COM(*n*) ON statement must be executed to allow trapping by the ON COM(*n*) statement. After COM(*n*) ON, if a non-zero line number is specified in the ON COM(*n*) statement, BASIC checks to see if any characters have come in to the communications adapter every time a new statement is executed.

If COM(*n*) is OFF, no trapping takes place and any communication activity is not remembered even if it does take place.

If a COM(*n*) STOP statement has been executed, no trapping can take place. However, any communications activity that does take place is remembered so that an immediate trap occurs when COM(*n*) ON is executed.

# COMMON Statement

---

**Purpose:** Passes variables to a chained program.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*                \*\*\*                (\*\*)

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

**Remarks:** *variable* is the name of a variable that is to be passed to the chained-to program. Array variables are specified by appending "( )" to the variable name.

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, although it is recommended that they appear at the beginning. Any number of COMMON statements may appear in a program, but the same variable cannot appear in more than one COMMON statement. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Any arrays that are passed do not need to be dimensioned in the chained-to program.

**Example:** 100 COMMON A,BEE1,C,D(),G\$  
110 CHAIN "A:PROG3"

This example chains to program PROG3 on the diskette in drive A:, and passes the array D along with the variables A, BEE1, C, and G\$.

# **CONT**

## **Command**

---

**Purpose:** Resumes program execution after a break.

**Versions:** Cassette      Disk      Advanced      Compiler

\*\*\*      \*\*\*      \*\*\*

**Format:** CONT

**Remarks:** The CONT command may be used to resume program execution after Ctrl-Break has been pressed, a STOP or END statement has been executed, or an error has occurred. Execution continues at the point where the break happened. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt.

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, you can examine or change the values of variables using direct mode statements. You may then use CONT to resume execution, or you may use a direct mode GOTO, which resumes execution at a particular line number.

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

# CONT Command

**Example:** In the following example, we create a long loop.

```
Ok
10 FOR A=1 TO 50
20 PRINT A;
30 NEXT A
RUN
 1  2  3  4  5  6  7  8  9  10 11 12
 13 14 15 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
```

(At this point we interrupt the loop by pressing Ctrl-Break.)

```
•
•
•
Break in 20
Ok
CONT
 30 31 32 33 34 35 36 37 38 39
 40 41 42 43 44 45 46 47 48 49
 50
Ok
```

# COS Function

---

**Purpose:** Returns the trigonometric cosine function.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{COS}(x)$

**Remarks:**  $x$  is the angle whose cosine is to be calculated. The value of  $x$  must be in radians. To convert from degrees to radians, multiply the degrees by PI/180, where PI=3.141593.

The calculation of COS( $x$ ) is performed in single precision.

**Example:** 0k  
10 PI=3.141593  
20 PRINT COS(PI)  
30 DEGREES=180  
40 RADIANS=DEGREES\*PI/180  
50 PRINT COS(RADIANS)  
RUN  
-1  
-1  
0k

This example shows, first, that the cosine of PI radians is equal to -1. Then it calculates the cosine of 180 degrees by first converting the degrees to radians (180 degrees happens to be the same as PI radians).

# CSNG Function

---

**Purpose:** Converts  $x$  to a single-precision number.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

**Format:**  $v = \text{CSNG}(x)$

**Remarks:**  $x$  is a numeric expression which will be converted to single-precision.

The rules outlined under “How BASIC Converts Numbers from One Precision to Another” in Chapter 3 are used for the conversion.

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

**Example:**

```
0k
10 A# = 975.3421222#
20 PRINT A#; CSNG(A#)
RUN
      975.3421222   975.3421
0k
```

The value of the double-precision number A# is rounded at the seventh digit and returned as CSNG(A#).

# CSRLIN Variable

---

**Purpose:** Returns the vertical coordinate of the cursor.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

**Format:**  $v = \text{CSRLIN}$

**Remarks:** The CSRLIN variable returns the current line (row) position of the cursor on the active page. (The active page is explained under "SCREEN Statement" in this chapter.) The value returned will be in the range 1 to 25.

The POS function returns the column location of the cursor. Refer to "POS Function" in this chapter.

Refer to "LOCATE Statement" to see how to set the cursor line.

**Example:**

```
10 Y = CSRLIN  'record current line
20 X = POS(0)  'record current column
29 'print HI MOM on line 24
30 LOCATE 24,1: PRINT "HI MOM"
40 LOCATE Y,X  'restore position
```

This example saves the cursor coordinates in the variables X and Y, then moves the cursor to line 24 to put the words "HI MOM" on that line. Then the cursor is moved back to its old position.

# CVI, CVS, CVD Functions

---

**Purpose:** Converts string variable types to numeric variable types.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*                \*\*\*                \*\*\*

**Format:**  $v = \text{CVI}(2\text{-byte string})$

$v = \text{CVS}(4\text{-byte string})$

$v = \text{CVD}(8\text{-byte string})$

**Remarks:** Numeric values that are read from a random file must be converted from strings into numbers. CVI converts a two-byte string to an integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

The CVI, CVS, and CVD functions do *not* change the bytes of the actual data. They only change the way BASIC interprets those bytes.

See also “MKI\$, MKS\$, MKD\$ Functions” in this chapter, and “Appendix B. BASIC Diskette Input and Output.”

**Example:** 70 FIELD #1,4 AS N\$, 12 AS B\$  
80 GET #1  
90 Y=CVS(N\$)

This example uses a random file (#1) which has fields defined as in line 70. Line 80 reads a record from the file. Line 90 uses the CVS function to interpret the first four bytes (N\$) of the record as a single-precision number. N\$ was probably originally a number which was written to the file using the MKS\$ function.

# DATA Statement

---

**Purpose:** Stores the numeric and string constants that are accessed by the program's READ statement(s).

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

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

**Remarks:** *constant* may be a numeric or string constant. No expressions are allowed in the list. The numeric constants may be in any format—integer, fixed point, floating point, hex, or octal. String constants in DATA statements do not need to be surrounded by quotation marks, unless the string contains commas, colons, or significant leading or trailing blanks.

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

# DATA Statement

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement or a “Syntax error” occurs.

You can use the RESTORE statement to reread information from any line in the list of DATA statements. (See “RESTORE Statement” in this chapter.)

**Example:** See examples under “READ Statement” in this chapter.

# **DATE\$**

## **Variable and Statement**

---

**Purpose:** Sets or retrieves the date.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*                 \*\*\*                 \*\*\*

**Format:** As a variable:

*v\$* = DATE\$

As a statement:

DATE\$ = *x\$*

**Remarks:** For the variable (*v\$* = DATE\$):

A 10-character string of the form *mm-dd-yyyy* is returned. Here, *mm* represents two digits for the month, *dd* is the day of the month (also 2 digits), and *yyyy* is the year. The date may have been set by DOS prior to entering BASIC.

For the statement (DATE\$ = *x\$*):

*x\$* is a string expression which is used to set the current date. You may enter *x\$* in any one of the following forms:

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

The year must be in the range 1980 to 2099. If you use only one digit for the month or day, a 0 (zero) is assumed in front of it. If you give only one digit for the year, a zero is appended to make it two digits. If you give only two digits for the year, the year is assumed to be 19*yy*.

# DATE\$

## Variable and Statement

**Example:** 0k

```
10 DATE$= "8/29/82"
20 PRINT DATE$
RUN
08-29-1982
0k
```

In the example we set the date to August 29th, 1982. Notice how, when we read the date back using the DATE\$ function, a zero was included in front of the month to make it two digits, and the year became 1982. Also, the month, day, and year are separated by hyphens even though we entered them as slashes.

# DEF FN Statement

---

**Purpose:** Defines and names a function that you write.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** DEF FN*name*[(*arg* [,*arg*]...)] =*expression*

**Remarks:** *name* is a valid variable name. This name, preceded by FN, becomes the name of the function.

*arg* is an argument. It is a variable name in the function definition that will be replaced with a value when the function is called. The arguments in the list represent, on a one-to-one basis, the values that are given when the function is called.

*expression* defines the returned value of the function. The type of the *expression* (numeric or string) must match the type declared by *name*.

The definition of the function is limited to one statement. Arguments (*arg*) that appear in the function definition serve only to define the function; they do not affect program variables that have the same name. A variable name used in the *expression* does not have to appear in the list of arguments. If it does, the value of the argument is supplied when the function is called. Otherwise, the current value of the variable is used.

# DEF FN Statement

The function type determines whether the function returns a numeric or string value. The type of the function is declared by *name*, in the same way as variables are declared (see “How to Declare Variable Types” in Chapter 3). If the type of *expression* (string or numeric) does not match the function type, a “Type mismatch” error occurs. If the function is numeric, the value of the expression is converted to the precision specified by *name* before it is returned to the calling statement.

A DEF FN statement must be executed to define a function before you may call that function. If a function is called before it has been defined, an “Undefined user function” error occurs. On the other hand, a function may be defined more than once. The most recently executed definition is used.

**Note:** You may have a *recursive* function, that is, one which calls itself. However, if you don’t provide a way to stop the recursion, an “Out of memory” error occurs.

DEF FN is invalid in direct mode.

**Example:** 0k

```
10 PI=3.141593
20 DEF FNAREA(R)=PI*R^2
30 INPUT "Radius? ",RADIUS
40 PRINT "Area is" FNAREA(RADIUS)
RUN
Radius?
```

(Suppose you respond with 2.)

```
Radius? 2
Area is 12.56637
0k
```

# **DEF FN Statement**

Line 20 defines the function FNAREA, which calculates the area of a circle with radius R. The function is called in line 40.

Here is an example with two arguments:

```
0k
10 DEF FNMUD(X,Y)=X-(INT(X/Y)*Y)
20 A = FNMUD(7.4,4)
30 PRINT A
RUN
3.4
0k
```

# DEF SEG Statement

---

**Purpose:** Defines the current “segment” of storage. A subsequent BLOAD, BSAVE, CALL, PEEK, POKE, or USR definition will define the actual physical address of its operation as an offset into this segment.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** DEF SEG [=address]

**Remarks:** *address* is a numeric expression in the range 0 to 65535.

The initial setting for the segment when BASIC is started is BASIC's Data Segment (DS). BASIC's Data Segment is the beginning of your user workspace in memory. If you execute a DEF SEG statement which changes the segment, the value does *not* get reset to BASIC's DS when you issue a RUN command.

If *address* is omitted from the DEF SEG statement, the segment is set to BASIC's Data Segment.

If *address* is given, it should be a value based upon a 16 byte boundary. The value is shifted left 4 bits (multiplied by 16) to form the segment address for the subsequent operation. That is, if *address* is in hexadecimal, a 0 (zero) is added to get the actual segment address. BASIC does not perform any checking to assure that the segment value is valid.

# **DEF SEG**

## **Statement**

DEF and SEG must be separated by a space. Otherwise, BASIC will interpret the statement **DEFSEG=100** to mean: "assign the value 100 to the variable DEFSEG."

Any value entered outside the range indicated will result in an "Illegal function call" error. The previous value will be retained.

Refer to "Appendix C. Machine Language Subroutines" for more information on using DEF SEG.

**Example:** 100 DEF SEG ' restore segment to BASIC DS

```
200 ' set segment to color screen buffer  
210 DEF SEG=&HB800
```

In the second example, the screen buffer for the Color/Graphics Monitor adapter is at absolute address B8000 hex. Since segments are specified on 16 byte boundaries, the last hex digit is dropped on the DEF SEG specification.

# DEFtype Statements

---

**Purpose:** Declares variable types as integer, single-precision, double-precision, or string.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            (\*\*)

**Format:** DEFtype *letter*[-*letter*] [,*letter* [-*letter*]]...

**Remarks:** *type*      is INT, SNG, DBL, or STR.

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

A DEFtype statement declares that the variable names beginning with the letter or letters specified will be that type of variable. However, a type declaration character (% , !, #, or \$) always takes precedence over a DEFtype statement in the typing of a variable. Refer to "How to Declare Variable Types" in Chapter 3.

If no type declaration statements are encountered, BASIC assumes that all variables without declaration characters are single-precision variables.

If type declaration statements are used, they should be at the beginning of the program. The DEFtype statement must be executed before you use any variables which it declares.

# **DEFtype Statements**

**Example:** Ok

```
10 DEFDBL L-P
20 DEFSTR A
30 DEFINT X,D-H
40 ORDER = 1#/3: PRINT ORDER
50 ANIMAL = "CAT": PRINT ANIMAL
60 X=10/3: PRINT X
RUN
.333333333333333
CAT
3
Ok
```

Line 10 declares that all variables beginning with the letter L, M, N, O, or P will be double-precision variables.

Line 20 causes all variables beginning with the letter A to be string variables.

Line 30 declares that all variables beginning with the letter X, D, E, F, G, or H will be integer variables.

# DEF USR Statement

---

**Purpose:** Specifies the starting address of a machine language subroutine, which is later called by the USR function.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** DEF USR[*n*]=*offset*

**Remarks:** *n* may be any digit from 0 to 9. It identifies the number of the USR routine whose address is being specified. If *n* is omitted, DEF USR0 is assumed.

*offset* is an integer expression in the range 0 to 65535. The value of *offset* is added to the current segment value to obtain the actual starting address of the USR routine. See "DEF SEG Statement" in this chapter.

It is possible to redefine the address for a USR routine. Any number of DEF USR statements may appear in a program, thus allowing access to as many subroutines as necessary. The most recently executed value is used for the offset.

Refer to "Appendix C. Machine Language Subroutines" for complete information.

**Example:** 200 DEF SEG = 0  
210 DEF USR0=24000  
500 X=USR0(Y+2)

This example calls a routine at absolute location 24000 in memory.

# **DELETE Command**

---

**Purpose:** Deletes program lines.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*

**Format:** DELETE [*line1*] [-*line2*]

**Remarks:** *line1*      is the line number of the first line to be deleted.

*line2*      is the line number of the last line to be deleted.

The DELETE command erases the specified range of lines from the program. BASIC always returns to command level after a DELETE is executed.

A period (.) may be used in place of the line number to indicate the current line. If you specify a line number which does not exist in the program, an "Illegal function call" error occurs.

**Example:** This example deletes line 40:

DELETE 40

The next example deletes line 40 through 100, inclusive:

DELETE 40-100

The last example deletes all lines up to and including line 40:

DELETE -40

# DIM Statement

---

**Purpose:** Specifies the maximum values for array variable subscripts and allocates storage accordingly.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                (\*\*\*)

**Format:** DIM *variable(subscripts)* [,*variable(subscripts)*]...

**Remarks:** *variable* is the name to be used for the array.

*subscripts* is a list of numeric expressions, separated by commas, which define the dimensions of the array.

When executed, the DIM statement sets all the elements of the specified numeric arrays to an initial value of zero. String array elements are all variable length, with an initial null value (zero length).

If an array variable name is used without a DIM statement, the maximum value of its subscript is assumed to be 10. If a subscript is used that is greater than the maximum specified, a “Subscript out of range” error occurs.

The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see “OPTION BASE Statement” in this chapter). The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. Both of these numbers are also limited by the size of memory and by the length of statements.

# DIM Statement

If you try to dimension an array more than once, a “Duplicate Definition” error occurs. You may, however, use the ERASE statement to erase an array so you can dimension it again. For more information about arrays, see “Arrays” in Chapter 3.

**Example:** Ok

```
10 WRRMAX=2
20 DIM SIS(12), WRR$(WRRMAX,2)
30 DATA 26.5, 37, 8,29,80, 9.9, EH800
40 DATA 7, 18, 55, 12, 5, 43
50 FOR I=0 TO 12
60 READ SIS(I)
70 NEXT I
80 DATA SHERRY, ROBERT, "A:"
90 DATA 'HI, SCOTT', HELLO, GOOD-BYE
100 DATA BOCA RATON, DELRAY, MIAMI
110 FOR I=0 TO 2: FOR J=0 TO 2
120 READ WRR$(I,J)
130 NEXT J,I
140 PRINT SIS(3); WRR$(2,0)
RUN
29 BOCA RATON
Ok
```

This example creates two arrays: a one-dimensional numeric array named SIS with 13 elements, SIS(0) through SIS(12); and a two-dimensional string array named WRR\$, with three rows and three columns.

# DRAW Statement

**Purpose:** Draws an object as specified by *string*.

**Versions:** Cassette    Disk    Advanced    Compiler  
                      \*\*\*                              (\*\*)

Graphics mode only.

**Format:** DRAW *string*

**Remarks:** You use the DRAW statement to draw using a *graphics definition language*. The language commands are contained in the string expression *string*. The string defines an object, which is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of *string* and interprets single letter commands from the contents of the string. These commands are detailed below:

The following movement commands begin movement from the last point referenced. After each command, the last point referenced is the last point the command draws.

**U n**    Move up.

**D n**    Move down.

**L n**    Move left.

**R n**    Move right.

**E n**    Move diagonally up and right.

**F n**    Move diagonally down and right.

**G n**    Move diagonally down and left.

**H n**    Move diagonally up and left.

# DRAW Statement

*n* in each of the preceding commands indicates the distance to move. The number of points moved is *n* times the scaling factor (set by the S command).

**M x,y** Move absolute or relative. If *x* has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

The aspect ratio of your screen determines the spacing of the horizontal, vertical, and diagonal points. For example, the standard aspect ratio of 4/3 indicates that the horizontal axis of the screen is 4/3 as long as the vertical axis. You can use this information to determine how many vertical points are equal in length to how many horizontal points.

For example, in medium resolution there are 320 horizontal points and 200 vertical points. That means 8 horizontal points are equal in length to 5 vertical points if the screen aspect ratio is 1/1. If the aspect ratio is different, you multiply the number of vertical points by the aspect ratio. For example, using the standard aspect ratio of 4/3, in medium resolution 8 horizontal points are equal in length to 20/3 vertical points, or 24 horizontal equal 20 vertical. That is:

```
DRAW "U80 R96 D80 L96"
```

produces a square in medium resolution. Following similar reasoning, again with the standard screen aspect ratio of 4/3, in high resolution 48 horizontal points are equal in length to 20 vertical points.

# DRAW Statement

The following two prefix commands may precede any of the above movement commands.

- B** Move, but don't plot any points.
- N** Move, but return to the original position when finished.

The following commands are also available:

- A n** Set angle  $n$ .  $n$  may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they appear the same size as with 0 or 180 degrees on a display screen with standard aspect ratio 4/3.
- C n** Set color  $n$ .  $n$  may range from 0 to 3 in medium resolution, and 0 to 1 in high resolution. In medium resolution,  $n$  selects the color from the current palette as defined by the COLOR statement. 0 is the background color. The default is the foreground color, color number 3. In high resolution,  $n$  equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.
- S n** Set scale factor.  $n$  may range from 1 to 255.  $n$  divided by 4 is the scale factor. For example, if  $n=1$ , then the scale factor is 1/4. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and relative M commands gives the actual distance moved. The default value is 4, so the scale factor is 1.
- X variable;** Execute substring. This allows you to execute a second string from within a string.

# DRAW Statement

In all of these commands, the *n*, *x*, or *y* argument can be a constant like 123 or it can be =*variable*; where *variable* is the name of a numeric variable. The semicolon ( ; ) is required when you use a variable this way, or in the X command. Otherwise, a semicolon is optional between commands. Spaces are ignored in *string*. For example, you could use variables in a move command this way:

```
M+=X1; ,-=X2;
```

You can also specify variables in the form VARPTR\$(*variable*), instead of =*variable*. This is useful in programs that will later be compiled. For example:

## One Method

```
DRAW "XA$;"  
DRAW "S=SCALE;"
```

## Alternative Method

```
DRAW "X"+VARPTR$(A$)  
DRAW "S)+"+VARPTR$(SCALE)
```

The X command can be a very useful part of DRAW, because you can define a part of an object separate from the entire object. For example, a leg could be part of a man. You can also use X to draw a string of commands more than 255 characters long.

When coordinates which are out of range are given to DRAW, the coordinate which is out of range is given the closest valid value. In other words, the negative values become zero and Y values greater than 199 become 199. X values greater than 639 become 639. X values greater than 319 in medium resolution wrap to the next horizontal line.

# DRAW Statement

**Example:** To draw a box:

```
5 SCREEN 1  
10 A=20  
20 DRAW "U=A;R=A;D=A;L=A;"
```

To draw a triangle:

```
10 SCREEN 1  
20 DRAW "E15 F15 L30"
```

To create a “shooting star:”

```
10 SCREEN 1,0: COLOR 0,0: CLS  
20 DRAW "BM300,25" initial point  
30 STAR$="M+7,17 M-17,-12 M+20,0 M-17,12 M+7,-17"  
40 FOR SCALE=1 TO 40 STEP 2  
50 DRAW "C1;S=SCALE; BM-2,0;XSTAR$;"  
60 NEXT
```

# EDIT Command

---

**Purpose:** Displays a line for editing.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*

**Format:** EDIT *line*

**Remarks:** *line* is the line number of a line existing in the program. If there is no such line, an “Undefined line number” error occurs. A period (.) can be used for the line number to refer to the current line.

The EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. The line may then be modified as described under “The BASIC Program Editor” in Chapter 2.

A period (.) can be used for the line number to refer to the current line. For example, if you have just entered a line and wish to go back and change it, the command EDIT . will redisplay the line for editing.

LIST may also be used to display program lines for changing. Refer to “LIST Command” in this chapter.

# END Statement

---

**Purpose:** Terminates program execution, closes all files, and returns to command level.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                (\*\*)

**Format:** END

**Remarks:** END statements may be placed anywhere in the program to terminate execution. END is different from STOP in two ways:

- END does not cause a “Break” message to be printed.
- END closes all files.

An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

**Example:** 520 IF K>1000 THEN END ELSE GOTO 20

This example ends the program if K is greater than 1000; otherwise, the program branches to line number 20.

# EOF Function

---

**Purpose:** Indicates an end of file condition.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

**Format:**  $v = \text{EOF}(filenum)$

**Remarks:** *filenum* is the number specified on the OPEN statement.

The EOF function is useful for avoiding an "Input past end" error. EOF returns -1 (true) if end of file has been reached on the specified file. A 0 (zero) is returned if end of file has not been reached.

EOF is meaningful only for a file opened for sequential input from diskette or cassette, or for a communications file. A -1 for a communications file means that the buffer is empty.

**Example:**

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

This example reads information from the sequential file named "DATA". Values are read into the array M until end of file is reached.

# ERASE Statement

---

**Purpose:** Eliminates arrays from a program.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

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

**Remarks:** *arrayname* is the name of an array you want to erase.

You might want to use the ERASE statement if you are running short of storage space while running your program. After arrays are erased, the space in memory which had been allocated for the arrays may be used for other purposes.

ERASE can also be used when you want to redimension arrays in your program. If you try to redimension an array without first erasing it, a "Duplicate Definition" error occurs.

The CLEAR command is used to erase *all* variables from the work area.

# ERASE Statement

**Example:** Ok  
10 START=FRE("")  
20 DIM BIG(100,100)  
30 MIDDLE=FRE("")  
40 ERASE BIG  
50 DIM BIG(10,10)  
60 FINAL=FRE("")  
70 PRINT START, MIDDLE, FINAL  
RUN  
62808                21980                62289  
Ok

This example uses the FRE function to illustrate how ERASE can be used to free memory. The array BIG used up about 40K-bytes of memory (62808-21980) when it was dimensioned as BIG(100,100). After it was erased, it could be redimensioned to BIG(10,10), and it only took up a little more than 500 bytes (62808-62289).

The actual values returned by the FRE function may be different on your computer.

# ERR and ERL Variables

---

**Purpose:** Return the error code and line number associated with an error.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{ERR}$

$v = \text{ERL}$

**Remarks:** The variable ERR contains the error code for the last error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error handling routine (refer to "ON ERROR Statement" in this chapter).

If you do test ERL in an IF...THEN statement, be sure to put the line number on the right side of the relational operator, like this:

IF ERL = *line number* THEN ...

The number must be on the right side of the operator for it to be renumbered by RENUM.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. Since you do not want this number to be changed during a RENUM, if you want to test whether an error occurred in a direct mode statement you should use the form:

IF 65535 = ERL THEN ...

# **ERR and ERL Variables**

ERR and ERL can be set using the ERROR statement (see next section).

BASIC error codes are listed in "Appendix A. Messages."

**Example:**

```
10 ON ERROR GOTO 100
20 LPRINT "This goes to the printer"
30 END
100 IF ERR=27 THEN LOCATE 23,1:
      PRINT "Check printer": RESUME
```

This example tests for a common problem: forgetting to put paper in the printer, or forgetting to switch it on.

# ERROR Statement

- 
- Purpose:** • Simulates the occurrence of a BASIC error; or  
• Allows you to define your own error codes.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** ERROR *n*

**Remarks:** *n* must be an integer expression between 0 and 255.

If the value of *n* is the same as an error code used by BASIC (see “Appendix A. Messages”), the ERROR statement simulates the occurrence of that error. If an error handling routine has been defined by the ON ERROR statement, the error routine is entered. Otherwise the error message corresponding to the code is displayed, and execution halts. (See first example below.)

To define your own error code, use a value that is different from any used by BASIC. (We suggest you use the highest available values; for example, values greater than 200.) This new error code may then be tested in an error handling routine, just like any other error. (See second example below.)

If you define your own code in this way, and you don’t handle it in an error handling routine, BASIC displays the message “Unprintable error,” and execution halts.

# ERROR Statement

**Example:** The first example simulates a “String too long” error.

```
Ok  
10 T = 15  
20 ERROR T  
RUN  
String too long in line 20  
Ok
```

The next example is a part of a game program that allows you to make bets. By using an error code of 210, which BASIC doesn't use, the program traps the error if you exceed the house limit.

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

# EXP Function

---

**Purpose:** Calculates the exponential function.

**Versions:** Cassette      Disk      Advanced      Compiler

\*\*\*

\*\*\*

\*\*\*

\*\*\*

**Format:**  $v = \text{EXP}(x)$

**Remarks:**  $x$  may be any numeric expression.

This function returns the mathematical number  $e$  raised to the  $x$  power.  $e$  is the base for natural logarithms. An overflow occurs if  $x$  is greater than 88.02969.

**Example:** 0k

1Ø X = 2

2Ø PRINT EXP(X-1)

RUN

2.718282

0k

STATEMENTS

This example calculates  $e$  raised to the (2-1) power, which is simply  $e$ .

# FIELD Statement

---

**Purpose:** Allocates space for variables in a random file buffer.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                    \*\*\*                    \*\*\*

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

**Remarks:** *filenum* is the number under which the file was opened.

*width* is a numeric expression specifying the number of character positions to be allocated to *stringvar*.

*stringvar* is a string variable which will be used for random file access.

A FIELD statement defines variables that are used to get data out of a random buffer after a GET or to enter data into the buffer for a PUT.

The statement:

FIELD 1, 20 AS N\$, 10 AS ID\$, 40 AS ADD\$

allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does *not* actually place any data into the random file buffer. This is done by the LSET and RSET statements (see "LSET and RSET Statements" in this chapter).

# FIELD Statement

FIELD does not "remove" data from the file either. Information is read from the file into the random file buffer with the GET (file) statement. Information is read from the buffer by simply referring to the variables defined in the FIELD statement.

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

Any number of FIELD statements may be executed for the same file number, and all FIELD statements that have been executed are in effect at the same time. Each new FIELD statement redefines the buffer from the first character position, so this has the effect of having multiple field definitions for the same data.

**Note:** *Be careful about using a fielded variable name in an input or assignment statement.* Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent input statement or LET statement with that variable name on the left side of the equal sign is executed, the variable is moved to string space and is no longer in the file buffer.

See "Appendix B. BASIC Diskette Input and Output" for a complete explanation of how to use random files.

# FIELD Statement

**Example:**

```
10 OPEN 'A:CUST' AS #1
20 FIELD 1, 2 AS CUSTNO$, 30 AS CUSTNAME$,
      35 AS ADDR$
30 LSET CUSTNAME$+"O'NEIL INC"
40 LSET ADDR$+"50 SE 12TH ST, NY, NY"
50 LSET CUSTNO$=MK1$(7850)
60 PUT 1,1
70 GET 1,1
80 CNUM% = CVI(CUSTNO$): N$ = CUSTNAME$
90 PRINT CNUM%, N$, ADDR$
```

This example opens a file named "CUST" as a random file. The variable CUSTNO\$ is assigned to the first 2 positions in each record, CUSTNAME\$ is assigned to the next 30 positions, and ADDR\$ is assigned to the next 35 positions. Lines 30 through 50 put information into the buffer, and the PUT statement in line 60 writes the buffer to the file. Line 70 reads back that same record, and line 90 displays the three fields. Note in line 80 that it is okay to use a variable name which was defined in a FIELD statement on the *right* side of an assignment statement.

# FILES Command

---

**Purpose:** Displays the names of files residing on a diskette.  
The FILES command in BASIC is similar to the DIR command in DOS.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*           \*\*\*           \*\*\*

**Format:** FILES [*filespec*]

**Remarks:** *filespec* is a string expression for the file specification as explained under "Naming Files" in Chapter 3. If *filespec* is omitted, all the files on the DOS default drive will be listed.

All files matching the filename are displayed. The filename may contain question marks (?). A question mark matches any character in the name or extension. An asterisk (\*) as the first character of the name or extension will match any name or any extension.

If a drive is specified as part of *filespec*, then files which match the specified filename on the diskette in that drive are listed. Otherwise, the DOS default drive is used.

# **FILES**

## **Command**

**Example:** FILES

This displays all files on the DOS default diskette drive.

FILES "\*.BAS"

This displays all files with an extension of .BAS on the DOS default diskette drive.

FILES "B:\*.\*"

This displays all files on drive B:.

FILES "TEST???.BAS"

This lists all files on the DOS default drive which have a filename beginning with TEST followed by two or less other characters, and an extension of .BAS.

# FIX Function

---

**Purpose:** Truncates  $x$  to an integer.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{FIX}(x)$

**Remarks:**  $x$  may be any numeric expression.

FIX strips all digits to the right of the decimal point and returns the value of the digits to the left of the decimal point.

The difference between FIX and INT is that FIX does not return the next lower number when  $x$  is negative.

See the INT and CINT functions, which also return integers.

STATEMENTS

**Example:** 0k  
PRINT FIX(45.67)  
45  
0k  
PRINT FIX(-2.89)  
-2  
0k

Note in the examples how FIX does *not* round the decimal part when it converts to an integer.

# FOR and NEXT Statements

---

**Purpose:** Performs a series of instructions in a loop a given number of times.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            (\*\*\*)

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

.

.

.

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

**Remarks:** *variable* is an integer or single-precision variable to be used as a counter.

*x* is a numeric expression which is the initial value of the counter.

*y* is a numeric expression which is the final value of the counter.

*z* is a numeric expression to be used as an increment.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by the STEP value (*z*). If you do not specify a value for *z*, the increment is assumed to be 1 (one). A check is performed to see if the value of the counter is now greater than the final value *y*. If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

# FOR and NEXT Statements

If the value of  $z$  is negative, the test is reversed. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if  $x$  is already greater than  $y$  when the STEP value is positive, or if  $x$  is less than  $y$  when the STEP value is negative. If  $z$  is zero, an infinite loop will be created unless you provide some way to set the counter greater than the final value.

Program performance will be improved if you use integer counters whenever possible.

## Nested Loops

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

STATEMENTS

A NEXT statement of the form:

`NEXT var1, var2, var3 ...`

is equivalent to the sequence of statements:

```
NEXT var1  
NEXT var2  
NEXT var3  
.  
.  
.
```

# FOR and NEXT Statements

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement matches the most recent FOR statement. If you are using nested FOR...NEXT loops, you should include the variable(s) on all the NEXT statements. It is a good idea to include the variables in order to avoid confusion; but it can be necessary if you do any branching out of nested loops. (However, using variable names on the NEXT statements will cause your program to execute somewhat slower.)

If a NEXT statement is encountered before its corresponding FOR statement, a “NEXT without FOR” error occurs.

**Example:** The first example shows a FOR...NEXT loop with a STEP value of 2.

```
0k
10 J=10: K=30
20 FOR I=1 TO J STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
1 40
3 50
5 60
7 70
9 80
0k
```

# FOR and NEXT Statements

In the next example, the loop does not execute because the initial value of the loop is more than the final value:

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

In the last example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (This is different from some other versions of BASIC, which set the initial value of the counter before setting the final value. In another BASIC the loop in this example might execute six times.)

```
0k  
10 I=5  
20 FOR I=1 TO I+5  
30 PRINT I;  
40 NEXT  
RUN  
1 2 3 4 5 6 7 8 9 10  
0k
```

# FRE Function

**Purpose:** Returns the number of bytes in memory that are not being used by BASIC. This number does not include the size of the reserved portion of the interpreter workarea (normally 2.5K to 4K-bytes).

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            (\*\*)

**Format:**  $v = \text{FRE}(x)$

$v = \text{FRE}(x\$)$

**Remarks:**  $x$  and  $x$$  are dummy arguments.

Since strings in BASIC can have variable lengths (each time you do an assignment to a string its length may change), strings are manipulated dynamically. For this reason, string space may become fragmented.

FRE with any string value causes a housecleaning before returning the number of free bytes.

*Housecleaning* is when BASIC collects all of its useful data and frees up unused areas of memory that were once used for strings. The data is compressed so you can continue until you really run out of space.

BASIC automatically does a housecleaning when it is running out of usable workarea. You might want to use  $\text{FRE}("")$  periodically to get shorter delays for each housecleaning. Be patient: housecleaning may take a while.

# FRE Function

CLEAR ,*n* sets the maximum number of bytes for the BASIC workspace. FRE returns the amount of free storage in the BASIC workspace. If nothing is in the workspace, then the value returned by FRE will be 2.5K to 4K-bytes (the size of the reserved interpreter workarea) smaller than the number of bytes set by CLEAR.

**Example:**

```
0k
PRINT FRE(0)
14542
0k
```

The actual value returned by FRE on your computer may differ from this example.

# GET Statement (Files)

---

**Purpose:** Reads a record from a random file into a random buffer.

**Versions:** Cassette      Disk      Advanced      Compiler  
                  \*\*\*           \*\*\*           \*\*\*

**Format:** GET [#]*filenum* [, *number*]

**Remarks:** *filenum* is the number under which the file was opened.

*number* is the number of the record to be read, in the range 1 to 32767. If *number* is omitted, the next record (after the last GET) is read into the buffer.

After a GET statement, INPUT #, LINE INPUT #, or references to variables defined in the FIELD statement may be used to read characters from the random file buffer. Refer to "Appendix B. BASIC Diskette Input and Output" for more complete information on using GET.

Because BASIC and DOS block as many records as possible in 512 byte sectors, the GET statement does not necessarily perform a physical read from the diskette.

GET may also be used for communications files. In this case *number* is the number of bytes to read from the communications buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM..." statement.

# GET Statement (Files)

**Example:**

```
10 OPEN "A:CUST" AS #1
20 FIELD 1, 30 AS CUSTNAME$, 30 AS ADDR$,
   35 AS CITY$
30 GET 1
40 PRINT CUSTNAME$, ADDR$, CITY$
```

This example opens the file “CUST” for random access, with fields defined in line 20. The GET statement on line 30 reads a record into the file buffer. Line 40 displays the information from the record that was read.

# GET Statement (Graphics)

---

**Purpose:** Reads points from an area of the screen.

**Versions:** Cassette      Disk      Advanced      Compiler  
                      \*\*\*                                    \*\*\*  
                      Graphics mode only.

**Format:** GET  $(x_1, y_1)-(x_2, y_2)$ ,*arrayname*

**Remarks:**  $(x_1, y_1)$ ,  $(x_2, y_2)$

are coordinates in either absolute or relative form. Refer to "Specifying Coordinates" under "Graphics Modes" in Chapter 3 for information on coordinates.

*arrayname* is the name of the array you want to hold the information.

GET reads the colors of the points within the specified rectangle into the array. The specified rectangle has points  $(x_1, y_1)$  and  $(x_2, y_2)$  as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the B option.)

GET and PUT can be used for high speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations which move bits onto (PUT) and off of (GET) the screen. Remember that PUT and GET are also used for random access files, but the syntax of these statements is different.

# GET Statement (Graphics)

The array is used simply as a place to hold the image and must be numeric; it may be any precision, however. The required size of the array, in bytes, is:

$$4 + \text{INT}((x * \text{bitsperpixel} + 7) / 8) * y$$

where  $x$  and  $y$  are the lengths of the horizontal and vertical sides of the rectangle, respectively. The value of *bitsperpixel* is 2 in medium resolution, and 1 in high resolution.

For example, suppose we want to use the GET statement to get a 10 by 12 image in medium resolution. The number of bytes required is  $4 + \text{INT}((10 * 2 + 7) / 8) * 12$ , or 40 bytes. The bytes per element of an array are:

- 2 for integer
- 4 for single-precision
- 8 for double-precision

Therefore, we could use an integer array with at least 20 elements.

The information from the screen is stored in the array as follows:

1. two bytes giving the x dimension in bits
2. two bytes giving the y dimension in bits
3. the data itself

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The x dimension is in element 0 of the array, and the y dimension is in element 1. Keep in mind, however, that integers are stored low byte first, then high byte; but the data is actually transferred high byte first, then low byte.

# GET Statement (Graphics)

The data for each row of points in the rectangle is left justified on a byte boundary, so if there are less than a multiple of eight bits stored, the rest of the byte will be filled with zeros.

PUT and GET work significantly faster in medium resolution when  $x1 \text{ MOD } 4$  is equal to zero, and in high resolution when  $x1 \text{ MOD } 8$  is equal to zero. This is a special case where the rectangle boundaries fall on the byte boundaries.

# GOSUB and RETURN Statements

---

**Purpose:** Branches to and returns from a subroutine.

<b>Versions:</b>	Cassette ***	Disk ***	Advanced ***	Compiler ***
------------------	-----------------	-------------	-----------------	-----------------

**Format:** GOSUB *line*

  .

  .

  .

*RETURN*

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

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

The RETURN statement causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, if you want to return from different points in the subroutine. Subroutines may appear anywhere in the program.

To prevent your program from accidentally entering a subroutine, you may want to put a STOP, END, or GOTO statement in front of the subroutine to direct program control around it.

Use ON...GOSUB to branch to different subroutines based on the result of an expression.

# GOSUB and RETURN Statements

**Example:** Ok

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

This example shows how a subroutine works. The GOSUB in line 10 calls the subroutine in line 40. So the program branches to line 40 and starts executing statements there until it sees the RETURN statement in line 70. At that point the program goes back to the statement after the subroutine call; that is, it returns to line 20. The END statement in line 30 prevents the subroutine from being performed a second time.

# GOTO Statement

---

**Purpose:** Branches unconditionally out of the normal program sequence to a specified line number.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** GOTO *line*

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

If *line* is the line number of an executable statement, that statement and those following are executed. If *line* refers to a non-executable statement (such as REM or DATA), the program continues at the first executable statement encountered after *line*.

The GOTO statement can be used in direct mode to re-enter a program at a desired point. This can be useful in debugging.

Use ON...GOTO to branch to different lines based on the result of an expression.

# GOTO Statement

**Example:** Ok

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

The GOTO statement in line 50 puts the program into an infinite loop, which is stopped when the program runs out of data in the DATA statement. (Notice how branching to the DATA statement did not add additional values to the internal data table.)

# HEX\$ Function

---

**Purpose:** Returns a string which represents the hexadecimal value of the decimal argument.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v\$ = \text{HEX\$}(n)$

**Remarks:**  $n$  is a numeric expression in the range -32768 to 65535.

If  $n$  is negative, the two's complement form is used. That is,  $\text{HEX\$}(-n)$  is the same as  $\text{HEX\$}(65536-n)$ .

See the OCT\$ function for octal conversion.

**Example:** The following example uses the HEX\$ function to figure the hexadecimal representation for the two decimal values which are entered.

```
0k
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
    32 DECIMAL IS 20 HEXADECIMAL
0k
RUN
? 1023
    1023 DECIMAL IS 3FF HEXADECIMAL
0k
```

STATEMENTS

# IF Statement

---

**Purpose:** Makes a decision regarding program flow based on the result of an expression.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** IF *expression* [,]THEN *clause* [ELSE *clause*]

IF *expression* [,]GOTO *line* [,]ELSE *clause*]

**Remarks:** *expression* may be any numeric expression.

*clause*      may be a BASIC statement or a sequence of statements (separated by colons); or it may be simply the number of a line to branch to.

*line*      is the line number of a line existing in the program.

If the *expression* is true (not zero), the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number.

If the result of *expression* is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement.

If you enter an IF...THEN statement in direct mode, and it directs control to a line number, then an "Undefined line number" error results unless you

# IF Statement

previously entered a line with the specified line number.

**Note:** When using IF to test equality for a value that is the result of a single- or double-precision computation, remember that the internal representation of the value may not be exact. (This is because single- and double-precision values are stored internally in floating point binary format.) Therefore, the test should be against the *range* over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
1F ABS (A-1.0)<1.0E-6 THEN ...
```

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

Also note that IF... THEN...ELSE is just one statement. That is, the ELSE clause cannot be a separate program line. For example:

```
1Ø IF A=B THEN X=4  
2Ø ELSE P=Q
```

is invalid. Instead, it should be:

```
1Ø IF A=B THEN X=4 ELSE P=Q
```

# IF Statement

**Nesting of IF Statements:** IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

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

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

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

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

**Example:** This statement gets record I if I is not zero:

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

In the next example, if I is between 10 and 20, DB is calculated and execution branches to line 300. If I is not in this range, the message "OUT OF RANGE" is printed. Note the use of two statements in the THEN clause.

```
100 IF (I>10) AND (I<20) THEN  
    DB=1982-1: GOTO 300  
    ELSE PRINT "OUT OF RANGE"
```

This next statement causes printed output to go to either the screen or the printer, depending on the value of a variable (IOFLAG). If IOFLAG is false (zero), output goes to the printer; otherwise, output goes to the screen:

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

# INKEY\$ Variable

---

**Purpose:** Reads a character from the keyboard.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** *v\$* = INKEY\$

**Remarks:** INKEY\$ only reads a single character, even if there are several characters waiting in the keyboard buffer. The returned value is a zero-, one-, or two-character string.

- A null string (length zero) indicates that no character is pending at the keyboard.
- A one-character string contains the actual character read from the keyboard.
- A two-character string indicates a special extended code. The first character will be hex 00. For a complete list of these codes, see "Appendix G. ASCII Character Codes."

You must assign the result of INKEY\$ to a string variable before using the character with any BASIC statement or function.

While INKEY\$ is being used, no characters are displayed on the screen and all characters are passed through to the program except for:

- Ctrl-Break, which stops the program
- Ctrl-Num Lock, which sends the system into a pause state
- Alt-Ctrl-Del, which does a System Reset
- PrtSc, which prints the screen

# INKEY\$ Variable

If you press Enter in response to INKEY\$, the carriage return character passes through to the program.

**Note:** To avoid complications on the input buffer in Cassette BASIC, you should execute:

```
DEF SEG: POKE 106,0
```

after INKEY\$ has received the last character you want from a soft key string. This POKE is not required in Disk or Advanced BASIC.

**Example:** The following section of a program stops the program until any key on the keyboard is pressed:

```
110 PRINT "Press any key to continue"  
120 A$=INKEY$: IF A$="" THEN 120
```

The next example shows program lines that could be used to test a two-character code being returned:

```
210 KB$=INKEY$  
220 IF LEN(KB$)=2 THEN KB$=RIGHT$(KB$,1)
```

# INP Function

---

**Purpose:** Returns the byte read from port  $n$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*                \*\*\*                \*\*\*                \*\*\*

**Format:**  $v = \text{INP}(n)$

**Remarks:**  $n$  must be in the range 0 to 65535.

INP is the complementary function to the OUT statement (see "OUT Statement" in this chapter).

INP performs the same function as the IN instruction in assembly language. Refer to the *IBM Personal Computer Technical Reference* manual for a description of valid port numbers (I/O addresses).

**Example:** 100 A=INP(255)

This instruction reads a byte from port 255 and assigns it to the variable A.

# INPUT Statement

---

**Purpose:** Receives input from the keyboard during program execution.

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** INPUT[;][“*prompt*”];] *variable*[,*variable*]...

**Remarks:** “*prompt*” is a string constant which will be used to prompt for the desired input.

*variable* is the name of the numeric or string variable or array element which will receive the input.

When the program sees an INPUT statement, it pauses and displays a question mark on the screen to indicate that it is waiting for data. If a “*prompt*” is included, the string is displayed before the question mark. You may then enter the required data from the keyboard.

You may use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT “ENTER BIRTHDATE”,B\$ prints the prompt without the question mark.

The data that you enter is assigned to the *variable(s)* given in the variable list. The data items you supply must be separated by commas, and the number of data items must be the same as the number of variables in the list.

The type of each data item that you enter must agree with the type specified by the variable name. (Strings entered in response to an INPUT statement need not be surrounded by quotation marks unless they

# INPUT Statement

contain commas or significant leading or trailing blanks.)

If you respond to INPUT with too many or too few items, or with the wrong type of value (letters instead of numbers, etc.), BASIC displays the message "?Redo from start". If a single variable is requested, you may simply press Enter to indicate the default values of 0 for numeric input or null for string input. However, if more than one variable is requested, pressing Enter will cause the "?Redo from start" message to be printed because too few items were entered. BASIC does not assign any of the input values to variables until you give an acceptable response.

In Disk and Advanced BASIC, if INPUT is immediately followed by a semicolon, then pressing Enter to input data does not produce a carriage return/line feed sequence on the screen. This means that the cursor remains on the same line as your response.

**Example:** 0k

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

In this example, the question mark displayed by the computer is a prompt to tell you it wants you to enter something. Suppose you enter a 5. The program continues:

```
. .
?
5
5 SQUARED IS 25
0k
```

# INPUT Statement

```
Ok  
10 PI=3.14  
20 INPUT "WHAT IS THE RADIUS";R  
30 A=PI*R^2  
40 PRINT "THE AREA OF THE CIRCLE IS";A  
50 END  
RUN  
WHAT IS THE RADIUS?
```

For this second example, a prompt was included in line 20, so this time the computer prompts with "WHAT IS THE RADIUS?" Suppose you respond with 7.4. The program continues:

```
.  
. .  
WHAT IS THE RADIUS? 7.4  
THE AREA OF THE CIRCLE IS 171.9464  
Ok
```

# INPUT # Statement

---

**Purpose:** Reads data items from a sequential device or file and assigns them to program variables.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

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

**Remarks:** *filenum* is the number used when the file was opened for input.

*variable* is the name of a variable that will have an item in the file assigned to it. It may be a string or numeric variable, or an array element.

The sequential file may reside on diskette or on cassette; it may be a sequential data stream from a communications adapter; or it may be the keyboard (KYBD:).

The type of data in the file must match the type specified by the variable name. Unlike INPUT, no question mark is displayed with INPUT #.

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

## **INPUT # Statement**

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

INPUT # can also be used with a random file.

**Example:** See "Appendix B. BASIC Diskette Input and Output."

# INPUT\$ Function

---

**Purpose:** Returns a string of *n* characters, read from the keyboard or from file number *filenum*.

**Versions:** Cassette      Disk      Advanced      Compiler  
      \*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:** *v\$* = INPUT\$(*n*, [*#*]*filenum*)

**Remarks:** *n* is the number of characters to be read from the file.

*filenum* is the file number used on the OPEN statement. If *filenum* is omitted, the keyboard is read.

If the keyboard is used for input, no characters will be displayed on the screen. All characters (including control characters) are passed through except Ctrl-Break, which is used to interrupt the execution of the INPUT\$ function. When responding to INPUT\$ from the keyboard, it is not necessary to press Enter.

The INPUT\$ function enables you to read characters from the keyboard which are significant to the BASIC program editor, such as Backspace (ASCII code 8). If you want to read these special characters, you should use INPUT\$ or INKEY\$ (*not* INPUT or LINE INPUT).

For communications files, the INPUT\$ function is preferred over the INPUT # and LINE INPUT # statements, since all ASCII characters may be significant in communications. Refer to "Appendix F. Communications."

# **INPUT\$ Function**

**Example:** The following program lists the contents of a sequential file in hexadecimal.

```
10 OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

The next example reads a single character from the keyboard in response to a question.

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

# INSTR Function

---

**Purpose:** Searches for the first occurrence of string  $y\$$  in  $x\$$  and returns the position at which the match is found. The optional offset  $n$  sets the position for starting the search in  $x\$$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{INSTR}([n,]x\$,y\$)$

**Remarks:**  $n$  is a numeric expression in the range 1 to 255.

$x\$, y\$$  may be string variables, string expressions or string constants.

If  $n > \text{LEN}(x\$)$ , or if  $x\$$  is null, or if  $y\$$  cannot be found, INSTR returns 0. If  $y\$$  is null, INSTR returns  $n$  (or 1 if  $n$  is not specified).

If  $n$  is out of range, an “Illegal function call” error will be returned.

STATEMENTS

**Example:** 0k

```
10 A$ = "ABCDEB"  
20 B$ = "B"  
30 PRINT INSTR(A$,B$); INSTR(4,A$,B$)  
RUN  
2 6  
0k
```

This example searches for the string “B” within the string “ABCDEB”. When the string is searched from the beginning, “B” is found at position 2; when the search starts at position 4, “B” is found at position 6.

# INT Function

---

**Purpose:** Returns the largest integer that is less than or equal to  $x$ .

**Versions:** Cassette      Disk      Advanced      Compiler  
\*\*\*            \*\*\*            \*\*\*            \*\*\*

**Format:**  $v = \text{INT}(x)$

**Remarks:**  $x$  is any numeric expression.

This is called the “floor” function in some other programming languages.

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

**Example:**

```
0k
PRINT INT(45.67)
 45
0k
PRINT INT(-2.89)
 -3
0k
```

This example shows how INT truncates positive integers, but rounds negative numbers upward (in a negative direction).