IBM

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

CHAPTER 1. THE VERSIONS OF BASIC ... 1-1
The Versions of BASIC 1-3
Cassette BASIC 1-4
Disk BASIC 1-5
Advanced BASIC 1-6

CHAPTER 2. HOW TO START AND USE BASIC 2-1
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

CHAPTER 3. GENERAL INFORMATION ABOUT PROGRAMMING IN BASIC 3-1
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

Numeric Expressions and Operators 3-21
- Arithmetic Operators 3-21
- Relational Operators 3-23
- Logical Operators 3-25
- Numeric Functions 3-29
- Order of Execution 3-29

String Expressions and Operators 3-31
- Concatenation 3-31
- String Functions 3-32

Input and Output 3-33
- Files 3-33
- Using the Screen 3-38
- Other I/O Features 3-44

CHAPTER 4. BASIC COMMANDS, STATEMENTS, FUNCTIONS, AND VARIABLES 4-1

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
- String Functions 4-21

ABS Function 4-23

ASC Function 4-24

ATN Function 4-25

AUTO Command 4-26

BEEP Statement 4-28

BLOAD Command 4-29

BSAVE Command 4-32

CALL Statement 4-34

CDBL Function 4-35

CHAIN Statement 4-36

CHR$ Function 4-38

CINT Function 4-40

CIRCLE Statement 4-41

CLEAR Command 4-44

CLOSE Statement 4-46

CLS Statement 4-48

COLOR Statement 4-49
- The COLOR Statement in Text Mode ... 4-49
- The COLOR Statement in Graphics Mode 4-54

COM(n) Statement 4-56

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
LPRINT and LPRINT USING
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
ON...GOSUB and ON...GOTO
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
PRINT # and PRINT # USING
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

APPENDIX A. MESSAGES A-5

APPENDIX B. BASIC DISKETTE INPUT AND OUTPUT B-1
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

APPENDIX C. MACHINE LANGUAGE SUBROUTINES C-1
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

APPENDIX D. CONVERTING PROGRAMS TO IBM PERSONAL COMPUTER BASIC ... D-1
    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

APPENDIX E. MATHEMATICAL FUNCTIONS E-1

APPENDIX F. COMMUNICATIONS F-1
Opening a Communications File F-1
Communication I/O F-1
An Example Program F-4
Operation of Control Signals F-6
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

APPENDIX G. ASCII CHARACTER CODES G-1
Extended Codes G-6

APPENDIX H. HEXADECIMAL CONVERSION TABLE H-1

APPENDIX I. TECHNICAL INFORMATION AND TIPS I-1
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

APPENDIX J. GLOSSARY J-1

INDEX X-1

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:

Versions:	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 ¿. 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.

```
BASICA 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.

```
BASICA 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:

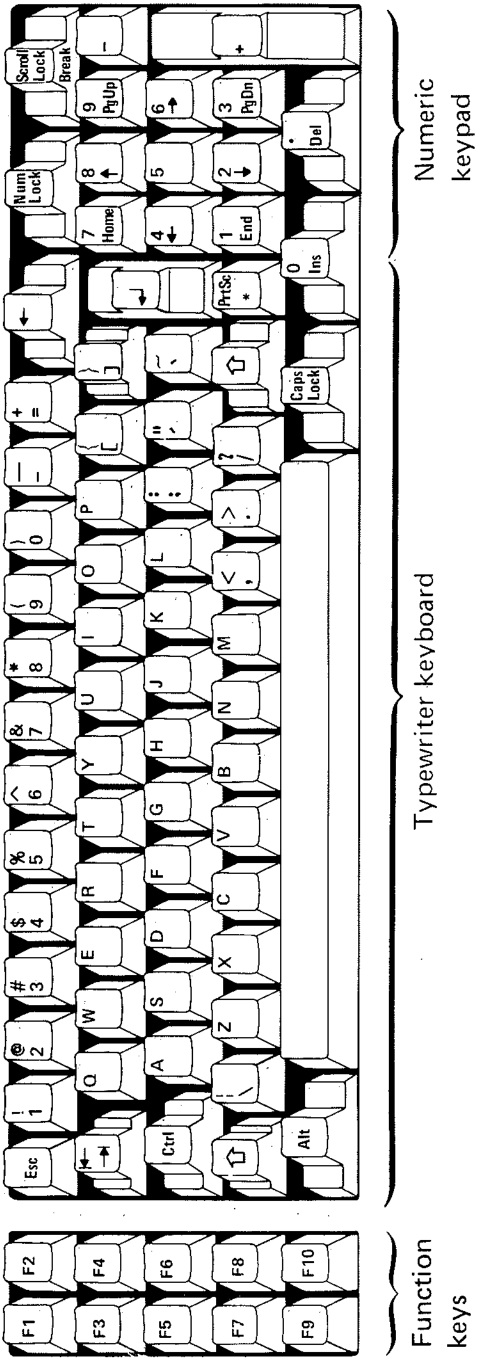
```
Ok
1 PRINT 20+2
RUN
 22
Ok
```

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

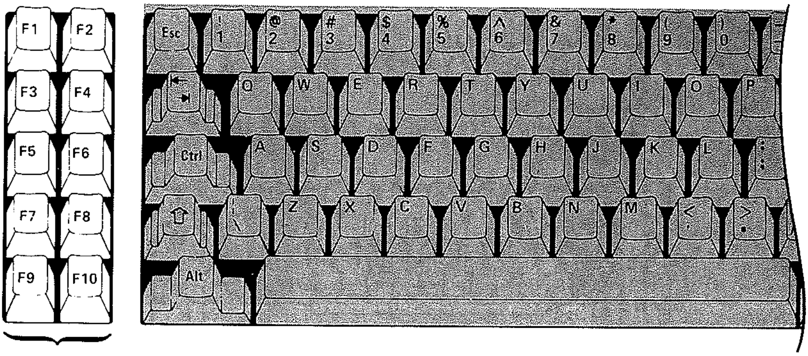

Esc
! 1
@ 2
# 3
$ 4
% 5
^ 6
& 7
* 8
(9
) 0
_ -
+ =
←
Num Lock
Scroll Lock Break
⇤ ⇥
Q
W
E
R
T
Y
U
I
O
P
{ [
}]
↵
7 Home
8 ↑
9 PgUp
-
Ctrl
A
S
D
F
G
H
J
K
L
: ;
" '
~ `
4 ←
5
6 →
⇧
| \
Z
X
C
V
B
N
M
< ,
> .
? /
⇧
PrtSc *
1 End
2 ↓
3 PgDn
+
Alt
Caps Lock
0 Ins
. Del
F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
Numeric keypad
Typewriter keyboard
Function keys

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:

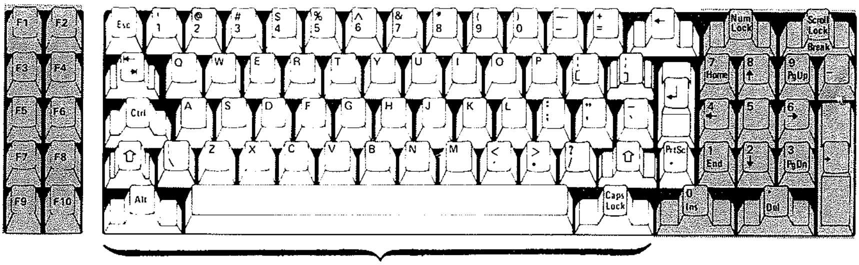
Function Keys



The function keys can be used:

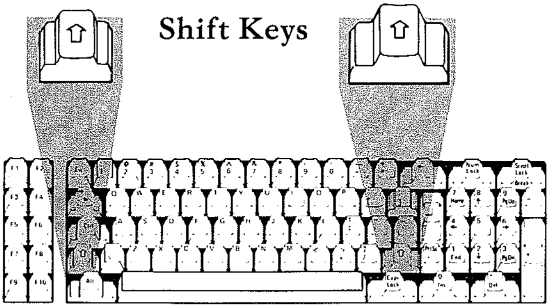
- As "soft keys." That is, you can set each key to automatically type any sequence of characters. In fact, some frequently-used commands have already been assigned to these keys. You may change these if you wish. Refer to "KEY Statement" in Chapter 4 for details.
- As program interrupts in Advanced BASIC,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

Typewriter Keyboard

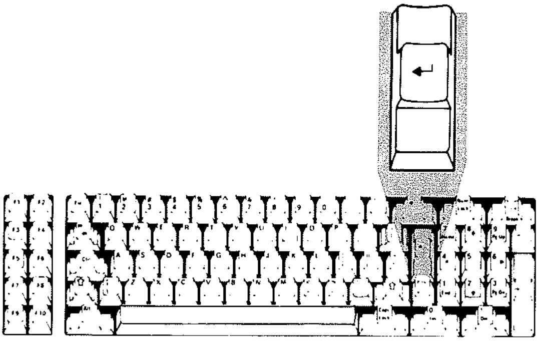

F1
F2
F3
F4
F5
F6
F7
F8
F9
F10
Esc
Num Lock
Scroll Lock Break
Ctrl
PrtSc
Alt
Caps Lock
Home
PgUp
End
PgDn
Ins
Del

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.



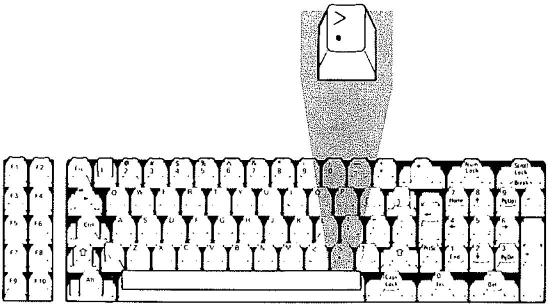
Capital letters and the special characters shown above the numbers on the number keys are displayed by holding down either of the Shift keys and pressing the desired key.



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.

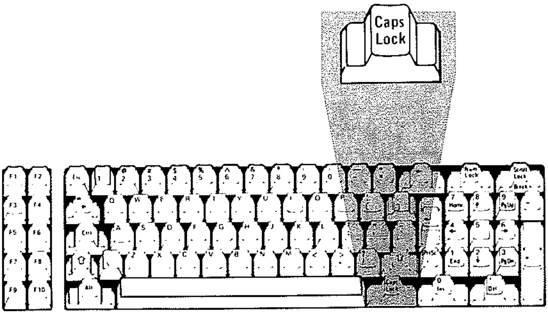
There are several important differences between this keyboard and a regular typewriter, however.

Special Symbols:



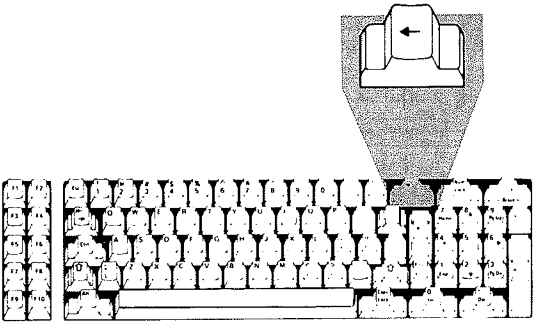
This keyboard has some special symbols that you won't find on a regular typewriter, like ^ , [, and]; and some characters are not where you might expect them to be if you're used to using a typewriter. For example, the uppershift period (.) is not a period, but the > symbol.

Uppercase:



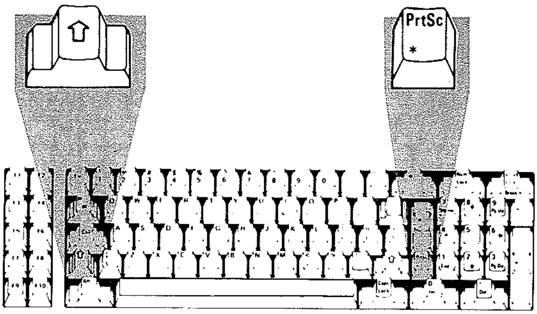
This keyboard does not have a normal Shift Lock key. The Caps Lock key is similar to a Shift Lock key, but it only gives you capital letters, and will not give you the uppershift characters on the numeric or other keys. After you press this key, you will continue to get capital letters until you press 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:



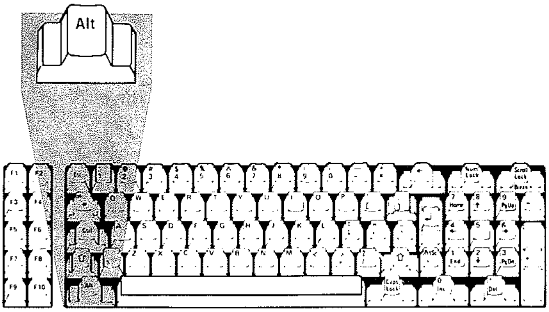
The Backspace key behaves somewhat differently from the Backspace key on a typewriter. It not only backspaces, it erases what you've typed as well. You should use the Cursor Left key to avoid erasing what you've typed. Refer to "The BASIC Program Editor" later in this chapter.

PrtSc:



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.

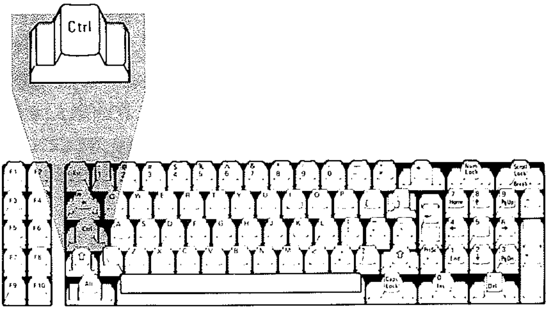


The Alt key enables easy entry of BASIC statement keywords. This key allows you to type an entire BASIC keyword with a single keystroke.

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.)

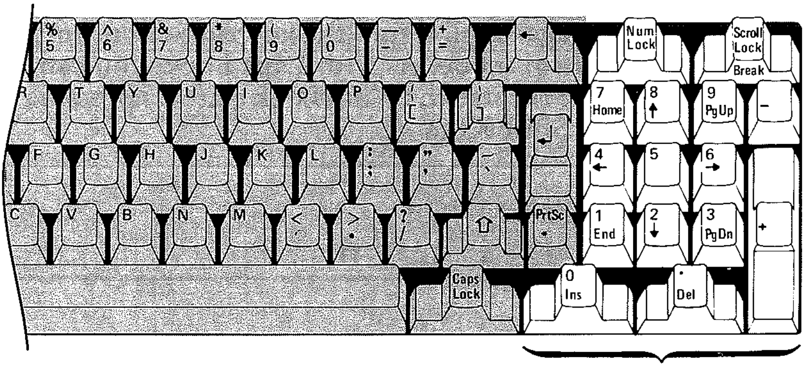


The Ctrl key is also used to enter certain codes and characters not otherwise available from the keyboard.

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.

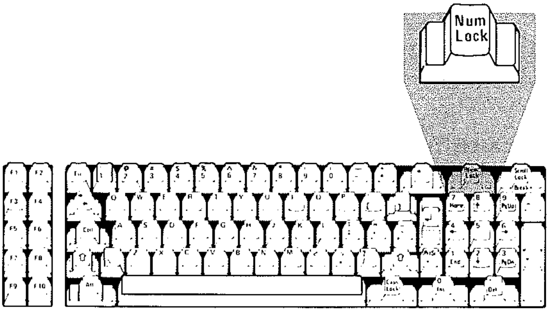
Numeric Keypad


Num Lock
Scroll Lock
Break
7 Home
8
9 PgUp
4
5
6
1 End
2
3 PgDn
0 Ins
Del
PrtSc
Caps Lock

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.

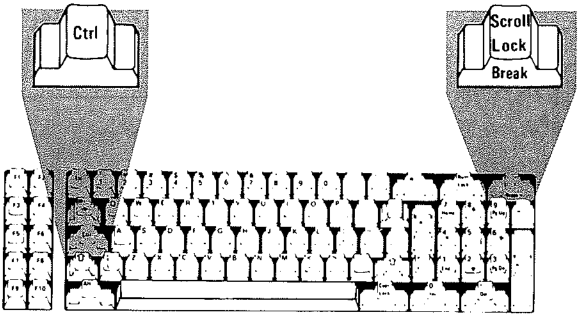


Keypad Shift: You can use the Num Lock key to set the numeric keypad so it works more like a calculator keypad. Pressing the Num Lock key shifts the numeric keypad into its own uppershift mode, so that you get the numbers 0 through 9 and the decimal point, as indicated on the keytops. Pressing Num Lock 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

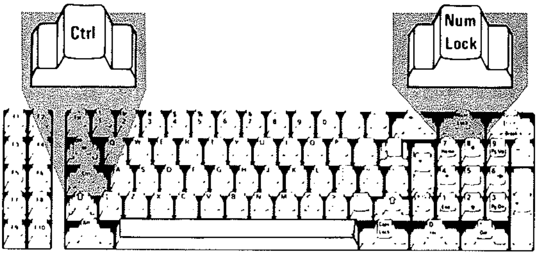
You should be aware of the special functions of the following combinations of keys:

Ctrl-Break



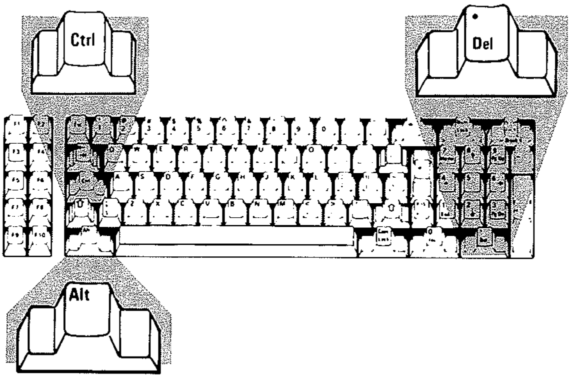
Ctrl-Break interrupts program execution at the next BASIC instruction and returns to BASIC command level. It is also used to exit AUTO line numbering mode.

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

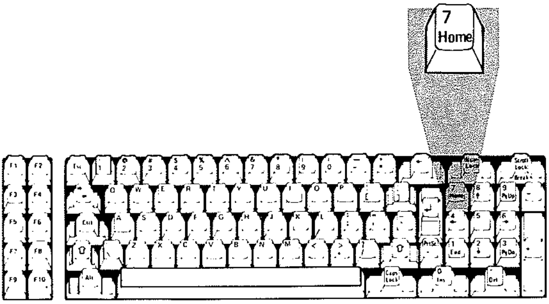
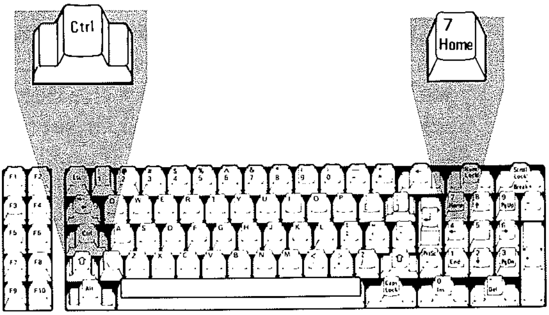
Any line of text typed while BASIC is at command level is processed by the BASIC program editor. The program editor is a "screen line editor." That is, you can change a line anywhere on the screen, but you can only change one line at a time. The change will only take effect if you press Enter on that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

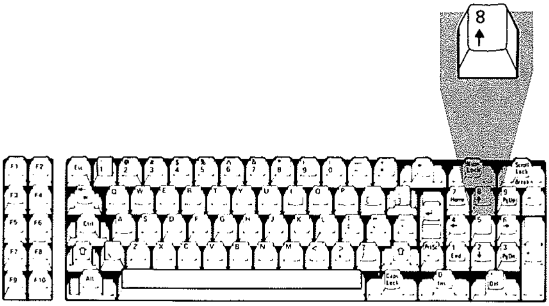
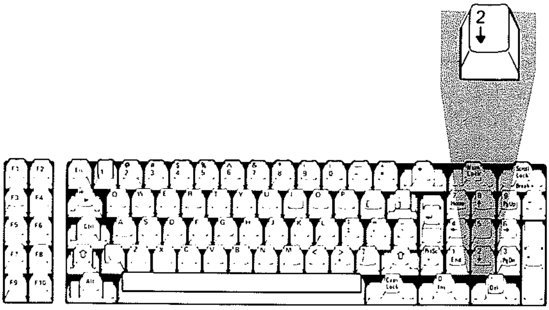
Use of the program editor can save a lot of time during program development. To become familiar with its features, we suggest you enter a sample program and practice all the editing capabilities. The best way for you to get a "feel" for the editing process is to try editing a few lines while studying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

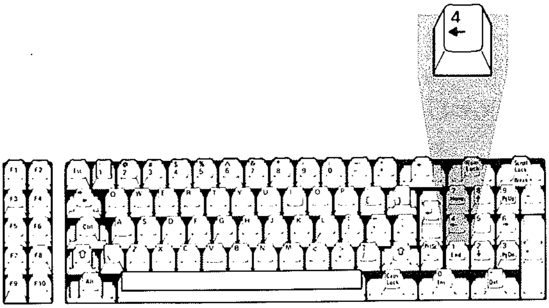
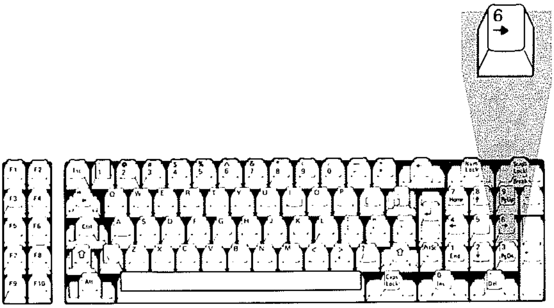
As you type things on your computer, you'll notice a blinking underline or box appearing just to the right of the last character you typed. This line or box is called the *cursor*. It marks the next position at which a character is to be typed, inserted, or deleted.

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
Home	 Moves the cursor to the upper left-hand corner of the screen.
Ctrl-Home	 Clears the screen and positions the cursor in the upper left-hand corner of the screen.

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

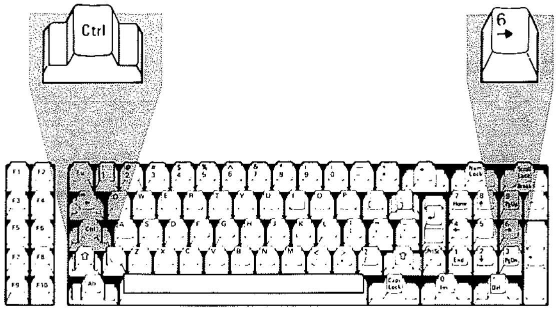
Key(s)	Function
← (Cursor Left)	 Moves the cursor one position left. If the cursor advances beyond the left edge of the screen, the cursor will move to the right side of the screen on the preceding line.
→ (Cursor Right)	 Moves the cursor one position right. If the cursor advances beyond the right edge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

<table>
<tr><th>Key(s)</th><th>Function</th></tr>
<tr><td>

Ctrl- →

(Next Word)

</td><td>



Moves the cursor right to the next *word*. 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.

For example, suppose we have the following line:

```
LINE (L1,LOW2)-(MAX,48) ,3 , BF
```

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:

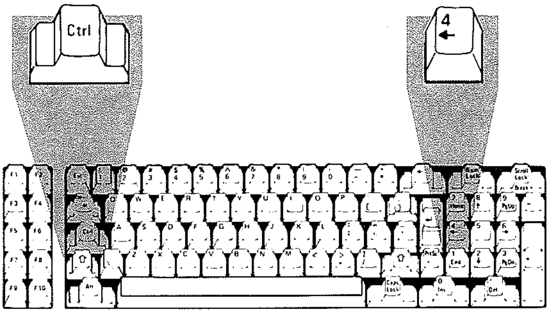
```
LINE (L1,LOW2)-(MAX,48) ,3 , BF
```

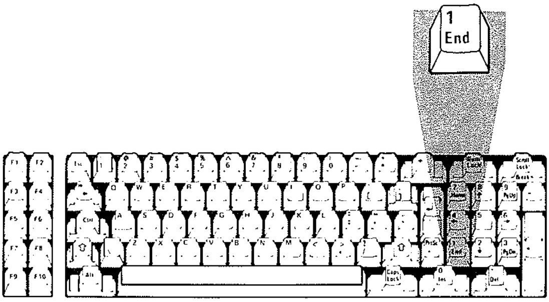
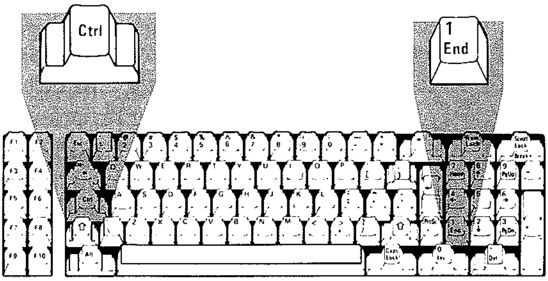
If we press Next Word again, the cursor will move to the next word, which is the number 48:

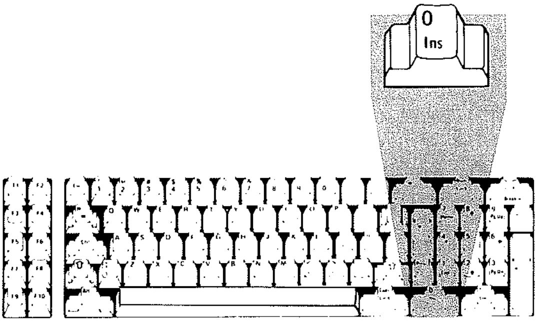
```
LINE (L1,LOW2)-(MAX,48) ,3 , BF
```

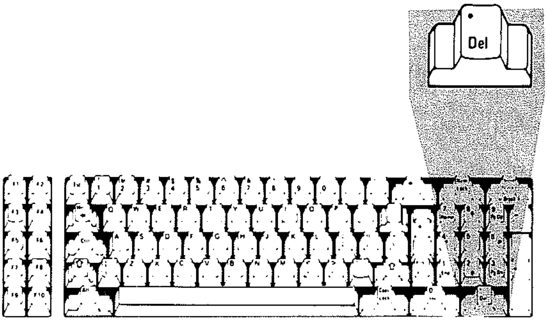
</td></tr>
</table>

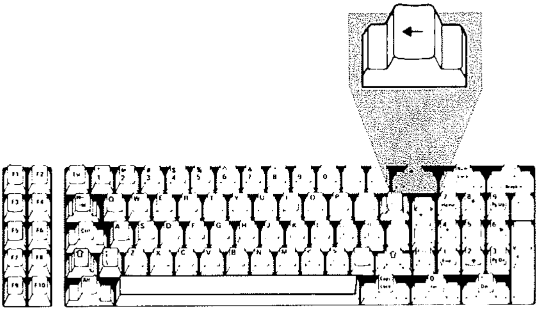
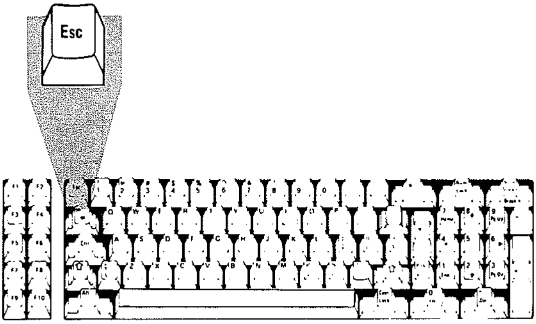
Key(s)	Function

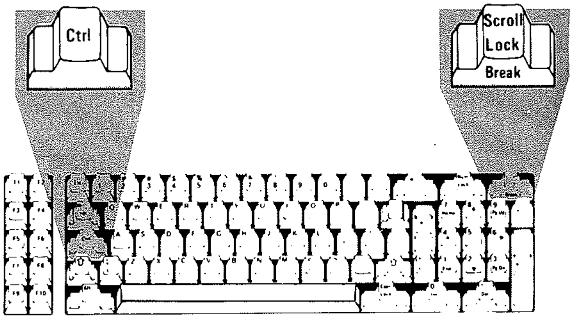
Ctrl- ←
(Previous Word)
	
	Moves the cursor left to the previous word. The previous word will be the letter or number to the left of the cursor which is preceded by a blank or special character.
	For example, suppose we have:
	`LINE (L1,LOW2)-(MAX,48) ,3 , BF_`
	If we press Previous Word (Ctrl-Cursor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

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

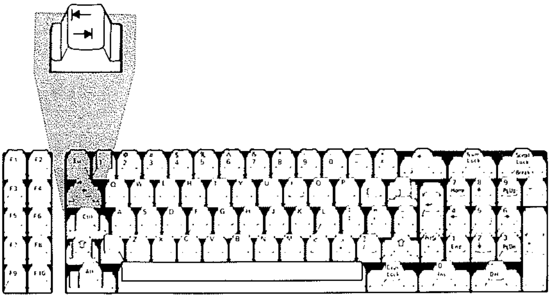
Key(s)	Function
Ins	 Sets insert mode. If insert mode is off, then pressing this key will turn it on. If insert mode is already on, then you will turn it off when you press this key. When you're in insert mode, the cursor covers the lower half of the character position. When insert mode is on, characters 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. When insert mode is off, any characters typed replace existing characters on the line. 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.

Key(s)	Function
Del	 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.

Key(s)	Function
← (Backspace)	 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.
Esc	 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.

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

Key(s)	Function

⇥
(Tab)
	
	Moves the cursor to the next tab stop. Tab stops occur every eight character positions; that is, at positions 1, 9, 17, etc.
	When insert mode is off, pressing the Tab key moves the cursor over characters until it reaches the next tab stop.
	For example, suppose we have the following line:
	`10 REM this is a remark`
	If we press the Tab key, the cursor will move to the ninth position as shown:
	`10 REM this is a remark`
	If we press the Tab key again, the cursor moves to the 17th position on the line:
	`10 REM this is a_remark`

Key(s)	Function

⇥
(Tab)
	(continued)
	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 **Ins**.
	For example, suppose we have this line:
	`10 REM this is a remark`
	If we press the Ins key and then the Tab key, blanks are inserted up to position 17:
	`10 REM th       is a remark`

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 accidently 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■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
10 A = 2$12
RUN
Syntax error in 10
Ok
10 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

String Expressions and Operators 3-31
  Concatenation 3-31
  String Functions 3-32

Input and Output 3-33
  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:

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

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

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

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

Variables

Variables are names used to represent values that are used in a BASIC program. 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,

```
10 EXP = 5
```

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

```
10 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

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

```
Ok
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
Ok
```

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:

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

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:

   ```
   Ok
   10 C = 55.8834567#
   20 PRINT C
   RUN
    55.88346
   Ok
   ```

   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:

```
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:

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

4. 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:

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

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

```
Ok
10 D = 6#/7
20 PRINT D
RUN
 .8571429
Ok
```

   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.

5. 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
*, /	Multiplication, Floating Point Division	X*Y X/Y
\	Integer Division	X\Y
MOD	Modulo Arithmetic	X 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:

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

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:

```
Ok
10 A = 7 MOD 4
20 PRINT A
RUN
 3
Ok
```

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

```
Ok
PRINT 25.68 MOD 6.99
 5
Ok
```

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
10 X=100
20 IF X <> 200 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<10
 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
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
$X(-Y)$	X*(-Y)

Note: 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:

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

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:ROTHERM.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

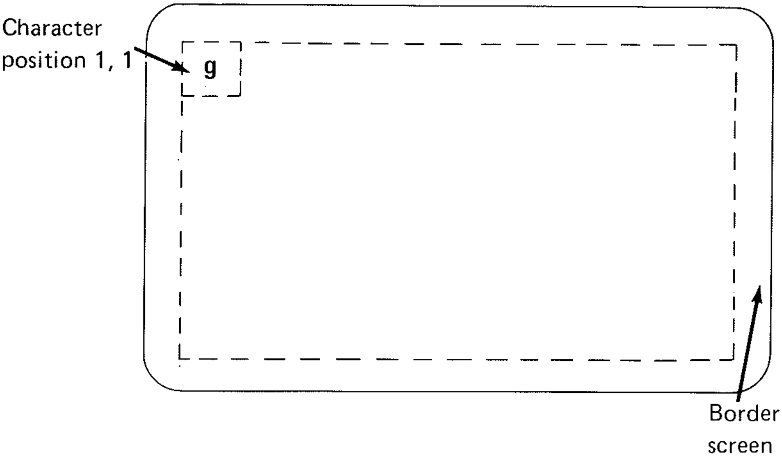
The IBM Personal Computer has two display adapters: the IBM Monochrome Display and Parallel Printer Adapter, and the Color/Graphics Monitor Adapter.

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
COLOR
LOCATE
PRINT
SCREEN
WIDTH
WRITE

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

CSRLIN
POS
SCREEN
SPC
TAB

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
COLOR
DRAW
GET
LINE
PAINT
PRESET
PSET
PUT
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 (*xoffset,yoffset*)

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

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

Command	Action
LIST line1-line2,filespec	Lists program lines on the screen or to the specified file.
LLIST line1-line2	Lists program lines on the printer.
LOAD filespec	Loads a program file. Can include the R option to run it.
MERGE filespec	Merges a saved program with the program in memory.
NAME filespec AS filename	Renames a diskette file.
NEW	Erases the current program and variables.
RENUM newnum,oldnum,increment	Renumbers program lines.
RESET	Reinitializes diskette information. Similar to CLOSE.
RUN filespec	Executes a program. The R option may be used to keep files open.
RUN line	Runs the program in memory starting at the specified line.
SAVE filespec	Saves the program in memory under the given filename. A or P option saves in ASCII or protected format.
SYSTEM	Ends BASIC. Closes all files and returns to DOS.
TRON, TROFF	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
CALL numvar(variable list)	Calls a machine language program.
CHAIN filespec	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.
COM(n) ON/OFF/STOP	Enables and disables trapping of communications activity.
COMMON list of variables	Passes variables to a chained program.
DATE$ = x$	Sets the date.
DEF FNname(arg list)=expression	Defines a numeric or string function.

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

Statement	Action
IF expression THEN clause ELSE clause	Performs the statement(s) in the THEN clause if expression is true (nonzero). Otherwise, performs the ELSE clause or goes to the next line.
KEY ON/OFF/LIST	Displays soft keys or turns display off.
KEY n, x$	Sets soft key n to the value of the string x$.
KEY(n) ON/OFF/STOP	Enables/disables trapping of function keys or cursor control keys.
LET variable=expression	Assigns the value of the expression to the variable.
MID$(v$,n,m)=y$	Replaces part of the variable v$ with the string y$, starting at position n and replacing m characters.
MOTOR state	Turns cassette motor on if state is nonzero, off if state is zero.
NEXT variable	Closes a FOR...NEXT loop (see FOR).
ON COM(n) GOSUB line	Enables trap routine for communications activity.
ON ERROR GOTO line	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
RETURN line	Returns from subroutine.
STOP	Stops program execution, prints a break message, and returns to command level.
STRIG ON/OFF	Enables/disables joystick button function.
STRIG(n) ON/OFF/STOP	Enables/disables joystick button trapping.
SWAP variable1,variable2	Exchanges values of two variables.
TIME$ = x$	Sets the time.
WAIT port,n,m	Suspends program execution until the specified port develops the specified bit pattern.
WEND	Closes a WHILE...WEND loop (see WHILE).
WHILE expression	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
INPUT “prompt”;variable list	Reads data from the keyboard.
INPUT #f,variable list	Reads data from file f.
LINE (x1,y1)-(x2,y2)	Draws a line on the screen. Other parameters allow you to draw a box, and fill the box in.
LINE INPUT “prompt”;stringvar	Reads an entire line from the keyboard, ignoring commas or other delimiters.
LINE INPUT #f,stringvar	Reads an entire line from a file.
LOCATE row,col	Positions the cursor. Other parameters allow you to define the size of the cursor and whether it is visible or not.
LPRINT list of expressions	Prints data on the printer.
LPRINT USING v$;list of expressions	Prints data on the printer using the format specified by v$.
LSET stringvar=x$	Left-justifies a string in a field.
OPEN filespec FOR mode AS #f	Opens the file for the mode specified. Another option sets the record length for random files.

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

Statement	Action
PUT (x,y),array,action	Writes graphic information to the screen.
READ variable list	Retrieves information from the data table created by DATA statements.
RSET stringvar=x$	Right-justifies a string in a field. See LSET.
SCREEN mode,burst,apage,vpage	Sets screen mode, color on or off, display page, and active page.
SOUND freq,duration	Generates sound through the speaker.
WIDTH size	Sets screen width. Other options allow you to specify the width of a printer or a communications file.
WRITE list of expressions	Outputs data on the screen.
WRITE #f, list of expressions	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**
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent (in radians) of x.
CDBL(x)	Converts x to a double-precision number.
CINT(x)	Converts x to an integer by rounding.
COS(x)	Returns the cosine of angle x, where x is in radians.
CSNG(x)	Converts x to a single-precision number.

Function	Result
EXP(x)	Raises *e* to the x power.
FIX(x)	Truncates x to an integer.
INT(x)	Returns the largest integer less than or equal to x.
LOG(x)	Returns the natural logorithm of x.
RND(x)	Returns a random number.
SGN(x)	Returns the sign of x.
SIN(x)	Returns the sine of angle x, where x is in radians.
SQR(x)	Returns the square root of x.
TAN(x)	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
ASC(x$)	Returns the ASCII code for the first character in x$.
CVI(x$), CVS(x$), CVD(x$)	Converts x$ to a number of the indicated precision.
INSTR(n,x$,y$)	Returns the position of first occurrence of y$ in x$ starting at position n.
LEN(x$)	Returns the length of x$.
VAL(x$)	Returns the numeric value of x$.

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: ● next record number of random file ● number of sectors read or written for sequential file ● number of characters in communications input buffer
LOF(f)	Returns the length of file f: ● number of bytes (in multiples of 128) in sequential or random file ● number of bytes free in communications input buffer
LPOS(n)	Returns the carriage position of the printer.

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

String Functions (return a string value)

GENERAL

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

I/O and MISCELLANEOUS

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

Function	Result
MKI$(x), MKS$(x), MKD$(x)	Converts x in indicated precision to proper length string.
OCT$(n)	Converts n to an octal string.
SPC(n)	Prints n spaces in a PRINT or LPRINT statement.
STR$(x)	Converts x to a string value.
TAB(n)	Tabs to position n in a PRINT or LPRINT statement.
TIME$	Returns the system time.
VARPTR$(v)	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 = ABS(x)

Remarks: *x* may be any numeric expression.

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

Example:

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

The absolute value of -35 is positive 35.

ASC
Function

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

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = 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:

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

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* = 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.

Example:

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

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

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)

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.

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* = 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:

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

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
```

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:

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

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.

```
Ok
KEY 1,"AUTO"+CHR$(13)
Ok
```

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
```

CINT
Function

Purpose: Converts *x* to an integer.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = 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:

```
Ok
PRINT CINT(45.67)
 46
Ok
PRINT CINT(-2.89)
-3
Ok
```

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
		***	***

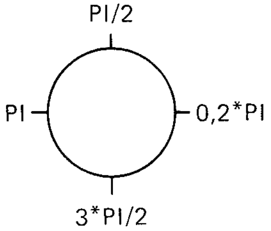
Graphics mode only.

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.

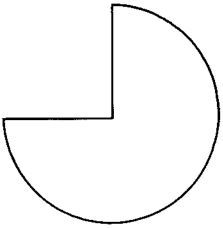
start and *end* specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:



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,

```
10 PI=3.141593
20 SCREEN 1
30 CIRCLE (160,100),60,,-PI,-PI/2
```

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

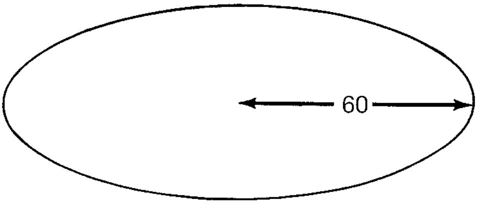


aspect affects the ratio of the x-radius to the y-radius. The default for *aspect* is 5/6 in medium resolution and 5/12 in high resolution. These values give a visual circle assuming the standard screen aspect ratio of 4/3.

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.

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*.

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 14,1,0
```

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.

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.

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* = 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:

```
Ok
10 PI=3.141593
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
RUN
-1
-1
Ok
```

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* = 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:

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

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* = 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* = CVI(*2-byte string*)

v = CVS(*4-byte string*)

v = CVD(8-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.

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*.

Example:

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

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.

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:

```
Ok
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
Ok
```

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:

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

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: DEF*type letter*[-*letter*] [,*letter* [-*letter*]]...

Remarks:

type	is INT, SNG, DBL, or STR.
letter	is a letter of the alphabet (A-Z).

A DEF*type* 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 DEF*type* 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 DEF*type* 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
 .3333333333333333
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$(WRRMAX,2)
30 DATA 26.5, 37, 8,29,80, 9.9, &H800
40 DATA 7, 18, 55, 12, 5, 43
50 FOR I=0 TO 12
60 READ SIS(I)
70 NEXT I
80 DATA SHERRY, ROBERT, "A:"
90 DATA "HI, SCOTT", HELLO, GOOD-BYE
100 DATA BOCA RATON, DELRAY, MIAMI
110 FOR I=0 TO 2: FOR J=0 TO 2
120 READ WRR$(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.

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.

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.

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	Alternative Method
`DRAW "XA$;"`	`DRAW "X"+VARPTR$(A$)`
`DRAW "S=SCALE;"`	`DRAW "S="+VARPTR$(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.

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* = 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.

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* = ERR

v = 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 = 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:

```
Ok
10 X = 2
20 PRINT EXP(X-1)
RUN
 2.718282
Ok
```

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 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$=MKI$(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* = 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.

Example:

```
Ok
PRINT FIX(45.67)
 45
Ok
PRINT FIX(-2.89)
-2
Ok
```

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.

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.

A NEXT statement of the form:

```
NEXT var1, var2, var3 ...
```

is equivalent to the sequence of statements:

```
NEXT var1
NEXT var2
NEXT var3
  .
  .
  .
```

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.

```
Ok
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
Ok
```

In the next example, the loop does not execute because the initial value of the loop is more than the final value:

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

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.)

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

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* = FRE(*x*)

v = 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 FRE("") periodically to get shorter delays for each housecleaning. Be patient: housecleaning may take a while.

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:

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

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.

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 (*x1,y1*)-(*x2,y2*),*arrayname*

Remarks: (*x1,y1*), (*x2,y2*)
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 (*x1,y1*) and (*x2,y2*) 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.

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+INT((*x***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+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.

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* MOD 4 is equal to zero, and in high resolution when *x1* 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.

Versions:

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$* = HEX$(*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, HEX$(-*n*) is the same as HEX$(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.

```
Ok
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
RUN
? 1023
 1023 DECIMAL IS 3FF HEXADECIMAL
Ok
```

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 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:
>
> ```
> IF 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:

```
10 IF A=B THEN X=4
20 ELSE P=Q
```

is invalid. Instead, it should be:

```
10 IF A=B THEN X=4 ELSE P=Q
```

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* = 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 queestion 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 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:

```
Ok
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
 Ok
```

```
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.

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* = 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*>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.

Example:

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

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* = 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:

```
Ok
PRINT INT(45.67)
 45
Ok
PRINT INT(-2.89)
-3
Ok
```

This example shows how INT truncates positive integers, but rounds negative numbers upward (in a negative direction).

KEY
Statement

Purpose: Sets or displays the soft keys.

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

Format: KEY *n, x$*

KEY LIST

KEY ON

KEY OFF

Remarks:

n	is the function key number in the range 1 to 10.
x$	is a string expression which will be assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The KEY statement allows function keys to be designated *soft keys*. That is, you can set each function key to automatically type any sequence of characters. A string of up to 15 characters may be assigned to any one or all of the ten function keys. When the key is pressed, the string will be input to BASIC.

Initially, the soft keys are assigned the following values:

F1	LIST	F2	RUN←
F3	LOAD“	F4	SAVE“
F5	CONT←	F6	,“LPT1:”←
F7	TRON←	F8	TROFF←
F9	KEY	F10	SCREEN 0,0,0←

The arrow (←) indicates Enter.

KEY
Statement

KEY ON causes the soft key values to be displayed on the 25th line. When the width is 40, five of the ten soft keys are displayed. When the width is 80, all ten are displayed. In either width, only the first six characters of each value are displayed. ON is the default state for the soft key display.

KEY OFF erases the soft key display from the 25th line, making that line available for program use. It does not disable the function keys.

KEY LIST lists all ten soft key values on the screen. All 15 characters of each value are displayed.

KEY *n, x$* assigns the value of *x$* to the function key specified (1 to 10). *x$* may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

If the value entered for *n* is not the range 1 to 10, an "Illegal function call" error occurs. The previous key string assignment is retained.

When a soft key is pressed, the INKEY$ function returns one character of the soft key string each time it is called. If the soft key is disabled, INKEY$ returns a two character string. The first character is binary zero, the second is the key scan code, as listed in "Appendix G. ASCII Character Codes."

Note: To avoid complications on the input buffer in Cassette BASIC, you should execute:

```
DEF SEG: POKE 106,0
```

after reassigning any soft keys and after INKEY$ has received the last character you want from a soft key string. This POKE is not required in Disk or Advanced BASIC.

After turning off the soft key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

See the following section, "KEY(n) Statement," to see how to enable and disable function key trapping in Advanced BASIC.

Example:

```
50 KEY ON
```

displays the soft keys on the 25th line.

```
200 KEY OFF
```

erases soft key display. The soft keys are still active, but not displayed.

```
10 KEY 1,"FILES"+CHR$(13)
```

assigns the string "FILES"+Enter to soft key 1. This is a way to assign a commonly used command to a function key.

```
20 KEY 1,""
```

disables function key 1 as a soft key.

KEY(n) Statement

Purpose: Activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement" in this chapter.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Remarks: *n* is a numeric expression in the range 1 to 14, and indicates the key to be trapped:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down

KEY(*n*) ON must be executed to activate trapping of function key or cursor control key activity. After KEY(*n*) ON, if a non-zero line number was specified in the ON KEY(*n*) statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY(*n*) statement.

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been executed, no trapping will take place. However, if you press the specified key your action is remembered so that an immediate trap takes place when KEY(*n*) ON is executed.

KEY (*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

If you use a KEY(*n*) statement in Cassette or Disk BASIC, you will get a "Syntax error." Refer to the previous section, "KEY Statement," for an explanation of the KEY statement without the (*n*).

KILL Command

Purpose: Deletes a file from a diskette. The KILL command in BASIC is similar to the ERASE command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: KILL *filespec*

Remarks:

filespec	is a valid file specification as explained under "Naming Files" in Chapter 3. The device name must be a diskette drive. If the device name is omitted, the DOS default drive is used.

KILL can be used for all types of diskette files. The name must include the extension, if one exists. For example, you may save a BASIC program using the command

```
SAVE "TEST"
```

BASIC supplies the extension **.BAS** for the SAVE command, but not for the KILL command. If you want to delete that program file later, you must say KILL "TEST.BAS", not KILL "TEST".

If a KILL statement is given for a file that is currently open, a "File already open" error occurs.

Example:

```
200 KILL "A:DATA1"
```

This example deletes the file named "DATA1" on drive A:.

LEFT$
Function

Purpose: Returns the leftmost *n* characters of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = LEFT$(*x$*,*n*)

Remarks:

x$	is any string expression.
n	is a numeric expression which must be in the range 0 to 255. It specifies the number of characters which are to be in the result.

If *n* is greater than LEN(*x$*), the entire string (*x$*) is returned. If *n*=0, the null string (length zero) is returned.

Also see the MID$ and RIGHT$ functions.

Example:

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

In this example, the LEFT$ function is used to extract the first five characters from the string "BASIC PROGRAM".

LEN
Function

Purpose: Returns the number of characters in *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = LEN(*x$*)

Remarks: *x$* is any string expression.

Unprintable characters and blanks are included in the count of the number of characters.

Example:

```
10 X$ = "BOCA RATON, FL"
20 PRINT LEN(X$)
RUN
 14
Ok
```

There are 14 characters in the string "BOCA RATON, FL", because the comma and the blank are counted.

LET
Statement

Purpose: Assigns the value of an expression to a variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: [LET] *variable=expression*

Remarks:

variable	is the name of the variable or array element which is to receive a value. It may be a string or numeric variable or array element.
expression	is the expression whose value will be assigned to *variable*. The type of the expression (string or numeric) must match the type of the variable, or a "Type mismatch" error will occur.

The word LET is optional, that is, the equal sign is sufficient when assigning an expression to a variable name.

LET
Statement

Example:

```
110 LET DORI=12
120 LET E=DORI+2
130 LET FDANCE$="HORA"
```

This example assigns the value 12 to the variable DORI. It then assigns the value 14, which is the value of the expression DORI+2, to the variable E. The string "HORA" is assigned to the variable FDANCE$.

The same statements could have also been written:

```
110 DORI= 12
120 E =DORI+2
130 FDANCE$ = "HORA"
```

LINE
Statement

Purpose: Draws a line or a box on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: LINE [(*x1,y1*)] -(*x2,y2*) [,[*color*] [,B[F]]]

Remarks: *(x1,y1), (x2,y2)*
are coordinates in either absolute or relative form. (See "Specifying Coordinates" under "Graphics Modes" in Chapter 3.)

color is the color number 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.

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This will draw a line from the last point referenced to the point (X2,Y2) in the foreground color.

We can include a starting point also:

```
LINE (0,0)-(319,199) 'diagonal down screen
LINE (0,100)-(319,100) 'bar across screen
```

We can indicate the color to draw the line in:

```
LINE (10,10)-(20,20),2 'draw in color 2

1 'draw random lines in random colors
10 SCREEN 1,0,0,0: CLS
20 LINE -(RND*319,RND*199),RND*4
30 GOTO 20

1 'alternating pattern - line on, line off
10 SCREEN 1,0,0,0: CLS
20 FOR X=0 TO 319
30 LINE (X,0)-(X,199),X AND 1
40 NEXT
```

The last argument to LINE is **B** - box, or **BF** - filled box. We can leave out *color* and include the final argument:

```
LINE (0,0)-(100,100),,B 'box in foreground
```

or we may include the color:

```
LINE (0,0)-(100,100),2,BF 'fill box color 2
```

The **B** tells BASIC to draw a rectangle with the points (*x1,y1*) and (*x2,y2*) as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

which perform the equivalent function.

The **BF** means draw the same rectangle as **B**, but also fill in the interior points with the selected color.

When coordinates which are out of range are given to the LINE statement, 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.

The last point referenced after a LINE statement is point (*x2,y2*). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example,

```
LINE (100,100)-STEP (10,-20)
```

will draw a line from (100,100) to (110,80).

Example: This example will draw random filled boxes in random colors.

```
10 CLS
20 SCREEN 1,0: COLOR 0,0
30 LINE -(RND*319,RND*199),RND*2+1,BF
40 GOTO 30 'boxes will overlap
```

LINE INPUT
Statement

Purpose: Reads an entire line (up to 254 characters) from the keyboard into a string variable, ignoring delimiters.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT[;][*"prompt"*;] *stringvar*

Remarks: *"prompt"* is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter is assigned to *stringvar*. Trailing blanks are ignored.

In Disk and Advanced BASIC, if LINE INPUT is immediately followed by a semicolon, then pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing Ctrl-Break. BASIC returns to command level and displays **Ok**. You may then enter CONT to resume execution at the LINE INPUT.

Example: See example in the next section, "LINE INPUT # Statement."

LINE INPUT
Statement

Purpose: Reads an entire line (up to 254 characters), ignoring delimiters, from a sequential file into a string variable.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LINE INPUT *#filenum, stringvar*

Remarks:

filenum	is the number under which the file was opened.
stringvar	is the name of a string variable or array element to which the line will be assigned.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT # reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved. That is, the line feed/carriage return characters are returned as part of the string.)

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

LINE INPUT # can also be used for random files. See "Appendix B. BASIC Diskette Input and Output."

LINE INPUT # Statement

Example: The following example uses LINE INPUT to get information from the keyboard, where the information is likely to have commas or other delimiters in it. Then the information is written to a sequential file, and read back out from the file using LINE INPUT #.

```
Ok
10 OPEN "LIST" FOR OUTPUT AS #1
20 LINE INPUT "Address? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "LIST" FOR INPUT AS #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN

Address?
```

Suppose you respond with DELRAY BEACH, FL 33445. The program continues:

```
●
●
●
Address? DELRAY BEACH, FL    33445

DELRAY BEACH, FL    33445
Ok
```

LIST
Command

Purpose: Lists the program currently in memory on the screen or other specified device.

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

Format: LIST [*line1*] [-[*line2*]] [,*filespec*]

Remarks: *line1, line2*

are valid line numbers in the range 0 to 65529. *line1* is the first line to be listed. *line2* is the last line to be listed. A period (.) may be used for either line number to indicate the current line.

filespec is a string expression for the file specification as outlined under "Naming Files" in Chapter 3. If *filespec* is omitted, the specified lines are listed on the screen.

In Cassette BASIC, listings directed to the screen by omitting the device specifier may be stopped at any time by pressing Ctrl-Break. Listings directed to specific devices may not be interrupted, and will list until the range is exhausted. That is, LIST *range* may be interrupted, but LIST *range*, "SCRN:" may not.

In Disk and Advanced BASIC, any listing to either the screen or the printer may be interrupted by pressing Ctrl-Break.

If the line range is omitted, the entire program is listed.

LIST Command

When the dash (-) is used in a line range, three options are available:

- If only *line1* is given, that line and all higher numbered lines are listed.
- If only *line2* is given, all lines from the beginning of the program through *line2* are listed.
- If both line numbers are specified, all lines from *line1* through *line2*, inclusive, are listed.

When you list to a file on cassette or diskette, the specified part of the program is saved in ASCII format. This file may later be used with MERGE.

Example:

```
LIST
```

Lists the entire program on the screen.

```
LIST 35,"SCRN:"
```

Lists line 35 on the screen.

```
LIST 10-20, "LPT1:"
```

Lists lines 10 through 20 on the printer.

```
LIST 100- ,"COM1:1200,N,8"
```

Lists all lines from 100 through the end of the program to the first communications adapter at 1200 bps, no parity, 8 data bits, 1 stop bit.

```
LIST -200,"CAS1:BOB"
```

Lists from the first line through line 200 to a file named "BOB" on cassette.

LLIST
Command

Purpose: Lists all or part of the program currently in memory on the printer (LPT1:).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

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

Remarks: The line number ranges for LLIST work the same as for LIST.

In Cassette BASIC, LLIST cannot be interrupted by Ctrl-Break. If you want to stop the list, you must turn the printer off.

BASIC always returns to command level after an LLIST is executed.

Example:

```
LLIST
```

Prints a listing of the entire program.

```
LLIST 35
```

Prints line 35.

```
LLIST 10-20
```

Lists lines 10 through 20 on the printer.

```
LLIST 100-
```

Prints all lines from 100 through the end of the program.

```
LLIST -200
```

Prints the first line through line 200.

LOAD Command

Purpose: Loads a program from the specified device into memory, and optionally runs it.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: LOAD *filespec*[,R]

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

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the specified program. If the **R** option is omitted, BASIC returns to direct mode after the program is loaded.

However, if the **R** option is used with LOAD, the program is run after it is loaded. In this case all open data files are kept open. Thus, LOAD with the **R** option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using data files.

LOAD *filespec*,R is equivalent to RUN *filespec*.

If you are using Cassette BASIC and the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for LOAD in Cassette BASIC.

LOAD Command

If you are using Disk or Advanced BASIC, the DOS default diskette drive is used if the device is omitted. The extension **.BAS** is added to the filename if no extension is supplied and the filename is eight characters or less.

Notes when using CAS1:

1. If the LOAD statement is entered 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 their corresponding 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)

   To see what files are on a cassette tape, rewind the tape and enter some name that is known not to be on the tape. For example, LOAD "CAS1:NOWHERE". All file names will then be displayed.

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

2. Note that Ctrl-Break may be typed at any time during LOAD. Between files or after a time-out period, BASIC will exit the search and return to command level. Previous memory contents remain unchanged.
3. If CAS1: is specified as the device and the filename is omitted, the next program file on the tape is loaded.

Example:

```
LOAD "MENU"
```

Loads the program named MENU, but does not run it.

```
LOAD "INVENT",R
```

Loads and runs the program INVENT.

```
RUN "INVENT"
```

Same as LOAD "INVENT",R.

```
LOAD "B:REPORT.BAS"
```

Loads the file REPORT.BAS from diskette drive B. Note that the .BAS did not have to be specified.

```
LOAD "CAS1:"
```

Loads the next program on the tape.

LOC
Function

Purpose: Returns the current position in the file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOC(*filenum*)

Remarks: *filenum* is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record is a 128 byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change this with the /C: option on the BASIC command. If there are more than 255 characters in the buffer, LOC returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for you to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

LOC
Function

Example:

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

This first example stops the program if we've gone past the 50th record in the file.

```
300 PUT #1,LOC(1)
```

The second example could be used to re-write the record that was just read.

LOCATE
Statement

Purpose: Positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: LOCATE [*row*][,[*col*] [,[*cursor*][,[*start*] [,*stop*]]]]

Remarks:

row	is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.
col	is a numeric expression in the range 1 to 40 or 1 to 80, depending upon screen width. It indicates the screen column number where you want to place the cursor.
cursor	is a value indicating whether the cursor is visible or not. A 0 (zero) indicates off, 1 (one) indicates on.
start	is the cursor starting scan line. It must be a numeric expression in the range 0 to 31.
stop	is the cursor stop scan line. It also must be numeric expression in the range 0 to 31.

cursor, start and *stop* do not apply to graphics mode.

start and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan lines. The scan lines are numbered from 0 at the top of the character position. The bottom scan line is 7 if you have the Color/Graphics Monitor Adapter, 13 if you have the IBM Monochrome Display and Parallel Printer Adapter. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is greater than *stop*, you'll get a two-part cursor. The cursor "wraps" from the bottom line back to the top.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, BASIC will not print to line 25. However, you can turn off the soft key display using KEY OFF, then use LOCATE 25,1: PRINT... to put things on line 25.

Any parameter may be omitted. Omitted parameters assume the current value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

Example:

```
10 LOCATE 1,1
```

Moves the cursor to the home position in the upper left-hand corner of the screen.

```
20 LOCATE ,,1
```

Makes the blinking cursor visible; its position remains unchanged.

```
30 LOCATE ,,,7
```

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character on the Color/Graphics Monitor Adapter (starting and ending on scan line 7).

```
40 LOCATE 5,1,1,0,7
```

Moves the cursor to line 5, column 1. Makes the cursor visible, covering the entire character cell on the Color/Graphics Monitor Adapter, starting at scan line 0 and ending on scan line 7.

LOF Function

Purpose: Returns the number of bytes allocated to the file (length of the file).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: *v* = LOF(*filenum*)

Remarks: *filenum* is the file number used when the file was opened.

For diskette files created by BASIC, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned. For diskette files created outside BASIC (for example, by using EDLIN), LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, *size*-LOC(*filenum*), where *size* is the size of the communications buffer, which defaults to 256 but may be changed with the /C: option on the BASIC command. Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in "Appendix F. Communications."

Example: These statements will get the last record of the file named BIG, assuming BIG was created with a record length of 128 bytes:

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

LOG
Function

Purpose: Returns the natural logarithm of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = LOG(*x*)

Remarks: *x* must be a numeric expression which is greater than zero.

The natural logarithm is the logarithm to the base *e*.

Example: The first example calculates the logarithm of the expression 45/7:

```
Ok
PRINT LOG(45/7)
 1.860752
Ok
```

The second example calculates the logorithm of *e* and of e^2 :

```
Ok
E= 2.718282
Ok
? LOG(E)
 1
Ok
? LOG(E*E)
 2
Ok
```

LPOS
Function

Purpose: Returns the current position of the print head within the printer buffer for LPT1:.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = LPOS(*n*)

Remarks: *n* is a numeric expression which is a dummy argument in Cassette BASIC. In Disk and Advanced BASIC, *n* indicates which printer is being tested, as follows:

0 or 1	LPT1:
2	LPT2:
3	LPT3:

Therefore, we recommend you use 0 or 1 in Cassette BASIC to maintain compatibility with the other versions.

The LPOS function does not necessarily give the physical position of the print head on the printer.

Example: In this example, if the line length is more than 60 characters long we send a carriage return character to the printer so it will skip to the next line.

```
100 IF LPOS(0)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Purpose:	Prints data on the printer (LPT1:).			
Versions:	Cassette ***	Disk ***	Advanced ***	Compiler ***

Format: LPRINT [*list of expressions*] [;]

LPRINT USING *v$; list of expressions* [;]

Remarks: *list of expressions*

is a list of the numeric and/or string expressions that are to be printed. The expressions must be separated by commas or semicolons.

v$ is a string constant or variable which identifies the format to be used for printing. This is explained in detail under "PRINT USING Statement."

These statements function like PRINT and PRINT USING, except output goes to the printer. See "PRINT Statement" and "PRINT USING Statement."

LPRINT assumes an 80-character wide printer. That is, BASIC automatically inserts a carriage return/line feed after printing 80 characters. This will result in two lines being skipped when you print exactly 80 characters, unless you end the statement with a semicolon. You may change the width value with a WIDTH "LPT1:" statement.

Printing is asynchronous with processing. If you do a form feed (LPRINT CHR$(12);) followed by another LPRINT and the printer takes more than 10 seconds to do the form feed, you may get a "Device

LPRINT and LPRINT USING Statements

Timeout" error on the second LPRINT. To avoid this problem, do the following:

```
1 ON ERROR GOTO 65000
  .
  .
  .
65000 IF ERR = 24 THEN RESUME '24=timeout
```

You might want to test ERL to make sure the timeout was caused by an LPRINT statement.

Example: This is an example of sending special control characters to the IBM 80 CPS Matrix Printer using LPRINT and CHR$. The printer control characters are listed in the *IBM Personal Computer Technical Reference* manual.

```
10 LPRINT CHR$(14);"      Title Line"
20 FOR I=2 TO 4
30 LPRINT "Report line";I
40 NEXT I
50 LPRINT CHR$(15);"Condensed print; 132 char/line"
60 LPRINT CHR$(18);"Return to normal"
70 LPRINT CHR$(27);"E"
80 LPRINT "This is emphasized print"
90 LPRINT CHR$(27);"F"
100 LPRINT "Back to normal again"
```

The output produced by this program looks like this:

```
                    T i t l e   L i n e
Report line 2
Report line 3
Report line 4
Condensed print; 132 char/line
Return to normal

This is emphasized print

Back to normal again
```

LSET and RSET Statements

Purpose: Moves data into a random file buffer (in preparation for a PUT (file) statement).

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: LSET *stringvar* = *x$*

RSET *stringvar* = *x$*

Remarks:

stringvar	is the name of a variable that was defined in a FIELD statement.
x$	is a string expression for the information to be placed into the field identified by *stringvar.*

If *x$* requires fewer bytes than were specified for *stringvar* in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If *x$* is longer than *stringvar*, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$ Functions" in this chapter.

LSET and RSET Statements

Refer to "Appendix B. BASIC Diskette Input and Output" for a complete explanation of using random files.

> **Note:** LSET or RSET may also be used with a string variable which was not defined in a FIELD statement to left-justify or right-justify a string in a given field. For example, the program lines:
>
> ```
> 110 A$=SPACE$(20)
> 120 RSET A$=N$
> ```
>
> right-justify the string N$ in a 20-character field. This can be useful for formatting printed output.

Example: This example converts the numeric value AMT into a string, and left-justifies it in the field A$ in preparation for a PUT (file) statement.

```
150 LSET A$=MKS$(AMT)
```

MERGE Command

Purpose: Merges the lines from an ASCII program file into the program currently in memory.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MERGE *filespec*

Remarks: *filespec* is a string expression for the file specification. It must conform to the rules for naming files as outlined in "Naming Files" in Chapter 3; otherwise an error occurs and the MERGE is cancelled.

The device is searched for the named file. If found, the program lines in the device file are merged with the lines in memory. If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file will replace the corresponding lines in memory.

After the MERGE command, the merged program resides in memory, and BASIC returns to command level.

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

If CAS1: is specified as the device name and the filename is omitted, the next ASCII program file encountered on the tape is merged.

MERGE Command

If the program being merged was not saved in ASCII format (using the A option on the SAVE command), a "Bad file mode" error occurs. The program in memory remains unchanged.

Example:

```
MERGE "A:NUMBRS"
```

This merges the file named "NUMBRS" on drive A: with the program in memory.

MID$
Function and Statement

Purpose: Returns the requested part of a given string. When used as a statement, as in the second format, replaces a portion of one string with another string.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a function:

*v$* = MID$(*x$*,*n*[,*m*])

As a statement:

MID$(*v$*,*n*[,*m*]) = *y$*

Remarks: **For the function (v$=MID$...):**

x$ is any string expression.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

The function returns a string of length *m* characters from *x$* beginning with the *n*th character. If *m* is omitted or if there are fewer than *m* characters to the right of the *n*th character, all rightmost characters beginning with the *n*th character are returned. If *m* is equal to zero, or if *n* is greater than LEN(*x$*), then MID$ returns a null string.

Also see the LEFT$ and RIGHT$ functions.

For the statement (MID$...=y$):

v$ is a string variable or array element that will have its characters replaced.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

y$ is a string expression.

The characters in *v$*, beginning at position *n*, are replaced by the characters in *y$*. The optional *m* refers to the number of characters from *y$* that will be used in the replacement. If *m* is omitted, all of *y$* is used.

However, regardless of whether *m* is omitted or included, the length of *v$* does not change. For example, if *v$* is four characters long and *y$* is five characters long, then after the replacement *v$* will contain only the first four characters of *y$*.

Note: If either *n* or *m* is out of range, an "Illegal function call" error will be returned.

MID$
Function and Statement

Example: The first example uses the MID$ function to select the middle portion of the string B$.

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

The next example uses the MID$ statement to replace characters in the string A$.

```
Ok
10 A$="MARATHON, GREECE"
20 MID$(A$,11)="FLORIDA"
30 PRINT A$
RUN
MARATHON, FLORID
Ok
```

Note in the second example how the length of A$ was not changed.

MKI$, MKS$, MKD$ Functions

Purpose: Convert numeric type values to string type values.

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

Format: *v$* = MKI$ (*integer expression*)

*v$* = MKS$ (*single-precision expression*)

*v$* = MKD$ (*double-precision expression*)

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

These functions differ from STR$ in that they do not actually change the bytes of the data, just the way BASIC interprets those bytes.

See also "CVI, CVS, CVD Functions" in this chapter and "Appendix B. BASIC Diskette Input and Output."

Example: This example uses a random file (#1) with fields defined in line 100. The first field, D$, is intended to hold a numeric value, AMT. Line 110 converts AMT to a string value using MKS$ and uses LSET to place what is actually the value of AMT into the random file buffer. Line 120 places a string into the buffer (we don't need to convert a string); then line 130 writes the data from the random file buffer to the file.

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

MOTOR
Statement

Purpose: Turns the cassette player on and off from a program.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: MOTOR [*state*]

Remarks: *state* is a numeric expression indicating on or off.

If *state* is non zero, the cassette motor is turned on. If *state* is zero, the cassette motor is turned off.

If *state* is omitted, the cassette motor state is switched. That is, if the motor is off, it is turned on and vice-versa.

Example: The following sequence of statements turns the cassette motor on, then off, then back on again.

```
10 MOTOR 1
20 MOTOR 0
30 MOTOR
```

NAME
Command

Purpose: Changes the name of a diskette file. The NAME command in BASIC is similar to the RENAME command in DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: NAME *filespec* AS *filename*

Remarks:

filespec	is a file specification as outlined under "Naming Files" in Chapter 3.
filename	will be the new filename. It must be a valid filename as outlined in the same section.

The file specified by *filespec* must exist and *filename* must not exist on the diskette, otherwise an error will result. If the device name is omitted, the DOS default drive is assumed. Note that the file extension does not default to .BAS.

After a NAME command, the file exists on the same diskette, in the same area of diskette space, with the new name.

Example:

```
NAME "A:ACCTS.BAS" AS "LEDGER.BAS"
```

In this example, the file that was formerly named ACCTS.BAS on the diskette in drive A will now be named LEDGER.BAS.

NEW
Command

Purpose: Deletes the program currently in memory and clears all variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	

Format: NEW

Remarks: NEW is usually used to free memory before entering a new program. BASIC always returns to command level after NEW is executed. NEW causes all files to be closed and turns trace off if it was on (see "TRON and TROFF Commands," later in this chapter).

Example:

```
Ok
NEW
Ok
```

The program that had been in memory is now deleted.

OCT$
Function

Purpose: Returns a string which represents the octal value of the decimal argument.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = OCT$(*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, OCT$(-*n*) is the same as OCT$(65536-*n*).

See the HEX$ function for hexadecimal conversion.

Example:

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

This example shows that 24 in decimal is 30 in octal.

ON COM(n) Statement

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

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

Format: ON COM(*n*) GOSUB *line*

Remarks:

n	is the number of the communications adapter (1 or 2).
line	is the line number of the beginning of the trap routine. Setting *line* equal to 0 (zero) disables trapping of communications activity for the specified adapter.

A COM(*n*) ON statement must be executed to activate this statement for adapter *n*. After COM(*n*) ON, if a non-zero line number is specified in the ON COM(*n*) statement then every time the program starts a new statement, BASIC checks to see if any characters have come in to the specified communications adapter. If so, BASIC performs a GOSUB to the specified *line*.

If COM(*n*) OFF is executed, no trapping takes place for the adapter. Even if communications activity does take place, the event is not remembered.

If a COM(*n*) STOP statement is executed, no trapping takes place for the adapter. However, any characters being received are remembered so an immediate trap takes place when COM(*n*) ON is executed.

When the trap occurs an automatic COM(*n*) STOP is executed so recursive traps can never take place.

The RETURN from the trap routine automatically does a COM(*n*) ON unless an explicit COM(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Typically the communications trap routine reads an entire message from the communications line before returning back. It is not recommended that you use the communications trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for communications to overflow.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
  .
  .
  .
500 REM incoming characters
  .
  .
  .
590 RETURN 300
```

This example sets up a trap routine for the first communications adapter at line 500.

ON ERROR
Statement

Purpose: Enables error trapping and specifies the first line of the error handling subroutine.

Versions:

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

Format: ON ERROR GOTO *line*

Remarks: *line* is the line number of the first line of the error trapping routine. If the line number does not exist, an "Undefined line number" error results.

Once error trapping has been enabled, all errors detected (*including direct mode errors*) will cause a jump to the specified error handling subroutine.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

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

You use the RESUME statement to exit from the error trapping routine. Refer to "RESUME Statement" in this chapter.

Example:

```
10 ON ERROR GOTO 100
20 LPRINT "This goes to the printer."
30 END
100 IF ERR=27 THEN PRINT "Check printer"
    : RESUME
```

This example shows how you might trap a common error — forgetting to put paper in the printer, or forgetting to switch it on.

ON...GOSUB and ON...GOTO Statements

Purpose: Branches to one of several specified line numbers, depending on the value of an expression.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: ON *n* GOTO *line*[,*line*]...

ON *n* GOSUB *line*[,*line*]...

Remarks:

n	is a numeric expression which is rounded to an integer, if necessary. It must be in the range 0 to 255, an "Illegal function call" error occurs.
line	is the line number of a line you wish to branch to.

The value of *n* determines which line number in the list will be used for branching. For example, if the value of *n* is 3, the third line number in the list will be the destination of the branch.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. That is, you eventually need to have a RETURN statement to bring you back to the line following the ON...GOSUB.

If the value of *n* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

ON...GOSUB and ON...GOTO Statements

Example: The first example branches to line 150 if L-1 equals 1, to line 300 if L-1 equals 2, to line 320 if L-1 equals 3, and to line 390 if L-1 equals 4. If L-1 is equal to 0 (zero) or is greater than 4, then the program just goes on to the next statement.

```
100 ON L-1 GOTO 150,300,320,390
```

The next example shows how to use an ON...GOSUB statement.

```
1200 ON A GOSUB 1300,1400
 .
 .
 .
1300 REM start of subroutine for A=1
 .
 .
 .
1390 RETURN
```

ON KEY(n) Statement

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

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

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON KEY(*n*) GOSUB *line*

Remarks: *n* is a numeric expression in the range 1 to 14 indicating the key to be trapped, as follows:

1-10	function keys F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to 0 disables trapping of the key.

A KEY(*n*) ON statement must be executed to activate this statement. After KEY(*n*) ON, if a non-zero line number is specified in the ON KEY(*n*) statement then every time the program starts a new statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line*.

If a KEY(*n*) OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is executed, no trapping takes place for the specified key. However, if the key is pressed the event is remembered, so an immediate trap takes place when KEY(*n*) ON is executed.

When the trap occurs an automatic KEY(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

ON KEY(n)
Statement

Example: The following is an example of a trap routine for function key 5.

```
100 ON KEY(5) GOSUB 200
110 KEY(5) ON
 .
 .
 .
200 REM function key 5 pressed
 .
 .
 .
290 RETURN 140
```

ON PEN
Statement

Purpose: Sets up a line number for BASIC to transfer control to when the light pen is activated.

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

Format: ON PEN GOSUB *line*

Remarks: *line* is the line number of the beginning of the trap routine for the light pen. Using a line number of 0 disables trapping of the light pen.

A PEN ON statement must be executed to activate this statement. After PEN ON, if a non-zero line number is specified in the ON PEN statement, then every time the program starts a new statement BASIC will check to see if the pen was activated. If so, BASIC performs a GOSUB *line*.

If PEN OFF is executed, no trapping takes place. Even if the light pen is activated, the event is not remembered.

If a PEN STOP statement is executed, no trapping takes place, but pen activity is remembered so that an immediate trap takes place when PEN ON is executed.

When the trap occurs, an automatic PEN STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a PEN ON unless an explicit PEN OFF was performed inside the trap routine.

ON PEN
Statement

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

PEN(0) is not set when pen activity causes a trap.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

> **Note:** Do not attempt any cassette I/O while PEN is ON.

Example: This example sets up a trap routine for the light pen.

```
10 ON PEN GOSUB 500
20 PEN ON
 .
 .
 .
500 REM subroutine for pen
 .
 .
 .
650 RETURN 30
```

ON STRIG(n)
Statement

Purpose: Sets up a line number for BASIC to trap to when one of the joystick buttons (triggers) is pressed.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: ON STRIG(*n*) GOSUB *line*

Remarks: *n* may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

line is the line number of the trapping routine. If *line* is 0, trapping of the joystick button is disabled.

A STRIG(*n*) ON statement must be executed to activate this statement for button *n*. If STRIG(*n*) ON is executed and a non-zero line number is specified in the ON STRIG(*n*) statement, then every time the program starts a new statement BASIC checks to see if the specified button has been pressed. If so, BASIC performs a GOSUB to the specified *line*.

If STRIG(*n*) OFF is executed, no trapping takes place for button *n*. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place for button *n*, but the button being pressed is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

When the trap occurs, an automatic STRIG(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a STRIG(*n*) ON unless an explicit STRIG(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Using STRIG(*n*) ON will activate the interrupt routine that checks the button status for the specified joystick button. Downstrokes that cause trapping will not set functions STRIG(0), STRIG(2), STRIG(4), or STRIG(6).

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Example: This is an example of a trapping routine for the button on the first joystick.

```
100 ON STRIG(0) GOSUB 2000
110 STRIG(0) ON
 .
 .
 .
2000 REM subroutine for 1st button
 .
 .
 .
2100 RETURN
```

OPEN Statement

Purpose: Allows I/O to a file or device.

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

Format: First form:

OPEN *filespec* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

Alternative form:

OPEN *mode2*, [#]*filenum, filespec* [,*recl*]

Remarks: *mode* in the first form, is one of the following:

OUTPUT	specifies sequential output mode.
INPUT	specifies sequential input mode.
APPEND	specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2 in the alternative form, is a string expression whose first character is one of the following:

O	specifies sequential output mode.
I	specifies sequential input mode.
R	specifies random input/output mode.

For both formats:

filenum	is an integer expression whose value is between one and the maximum number of files allowed. In Cassette BASIC, the maximum number is 4. In Disk and Advanced BASIC, the default maximum is 3, but this can be changed with the **/F:** option on the BASIC command.
filespec	is a string expression for the file specification as explained under "Naming Files" in Chapter 3.
recl	is an integer expression which, if included, sets the record length for random files. It may range from 1 to 32767. *recl* is not valid for sequential files. The default record length is 128 bytes. *recl* may not exceed the value set by the **/S:** option on the BASIC command.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

filenum is the number that is associated with the file for as long as it is open and is used by other I/O statements to refer to the file or device.

An OPEN must be executed before any I/O may be done to a device or file using any of the following statements, or any statement or function requiring a file number:

PRINT #
PRINT # USING
WRITE #
INPUT$
INPUT #
LINE INPUT #
GET
PUT

GET and PUT are valid for random files (or communications files — see next section). A diskette file may be either random or sequential, and a printer may be opened in either random or sequential mode; however, all other devices may be opened only for sequential operations.

BASIC normally adds a line feed after each carriage return (CHR$(13)) sent to a printer. However, if you open a printer (LPT1:, LPT2:, or LPT3:) as a random file with width 255, this line feed is suppressed.

APPEND is valid only for diskette files. The file pointer is initially set to the end of the file and the record number is set to the last record of the file. PRINT # or WRITE # will then extend the file.

> **Note:** At any one time, it is possible to have a particular file open under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, you may use different file numbers for different modes of access. Each file number has a different buffer, so you should use care if you are writing using one file number and reading using another file number.
>
> However, a file cannot be opened for sequential output or append if the file is already open.

If the device name is omitted when you are using Cassette BASIC, CAS1: is assumed. If you are using Disk or Advanced BASIC, the DOS default drive is assumed.

If CAS1: is specified as the device and the filename is omitted, then the next data file on the cassette is opened.

OPEN Statement

In Cassette BASIC, a maximum of four files may be open at one time (cassette, printer, keyboard, and screen). Note that only one cassette file may be open at a time. For Disk and Advanced BASIC the default maximum is three files. You can override this value by using the /F: option on the BASIC command.

If a file opened for input does not exist, a "File not found" error occurs. If a file which does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated will result in an "Illegal function call" error. The file is not opened.

See "Appendix B. BASIC Diskette Input and Output" for a complete explanation of using diskette files. Refer to the next section, "OPEN "COM... Statement," for information on opening communications files.

Example:

```
10 OPEN "DATA" FOR OUTPUT AS #1
```

or

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

Either of these statements opens the file named "DATA" for sequential output on the default device (CAS1: for Cassette BASIC, default drive for Disk and Advanced BASIC). Note that opening for output destroys any existing data in the file. If you do not wish to destroy data you should open for APPEND.

```
20 OPEN "B:SSFILE" AS 1 LEN=256
```

or

```
20 OPEN "R",1,"B:SSFILE",256
```

Either of the preceding two statements opens the file named "SSFILE" on the diskette in drive B for random input and output. The record length is 256.

```
25 FILE$ = "A:DATA.ART"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.ART" on the diskette in drive A and positions the file pointers so that any output to the file is placed at the end of existing data in the file.

```
Ok
10 OPEN "LPT1:" AS #1' random access
20 PRINT #1,"Printing width 80"
30 PRINT #1,"Now change to width 255"
40 WIDTH #1,255
50 PRINT #1,"This line will be underlined"
60 WIDTH #1,80
70 PRINT #1, STRING$(28,"_")
80 PRINT #1,"Printing width 80 with CR/LF"
RUN
Printing width 80
Now change to width 255
This line will be underlined
____________________________
Printing width 80 with CR/LF
Ok
```

Line 10 in this example opens the printer in random mode. Because the default width is 80, the lines printed by lines 20 and 30 end with a carriage return/line feed. Line 40 changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 50 ends only with a carriage return and not a line feed. This causes the line printed by line 70 to overprint "This line will be underlined", causing the line to be underlined. Line 60 changes the width back to 80 so the underlines and following lines will end with a line feed.

OPEN "COM... Statement

Purpose: Opens a communications file.

Versions:

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

Valid only with Asynchronous Communications Adapter.

Format: OPEN "COM*n*:[*speed*] [,*parity*] [,*data*] [,*stop*]
[,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF]"
AS [#]*filenum* [LEN=*number*]

Remarks: *n* is 1 or 2, indicating the number of the Asynchronous Communications Adapter.

speed is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default is 300 bps.

parity is a one-character constant specifying the parity for transmit and receive as follows:

S SPACE: Parity bit always transmitted and received as a space (0 bit).

O ODD: Odd transmit parity, odd receive parity checking.

M MARK: Parity bit always transmitted and received as a mark (1 bit).

E	EVEN: Even transmit parity, even receive parity checking.
N	NONE: No transmit parity, no receive parity checking.

The default is EVEN (E).

data	is an integer constant indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8. The default is 7.
stop	is an integer constant indicating the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 110 bps, one stop bit for all others. If you use 4 or 5 for *data*, a 2 here will mean 1 1/2 stop bits.
filenum	is an integer expression which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.
number	is the maximum number of bytes which can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. It supports RS232 asynchronous communication with other computers and peripherals.

A communications device may be open to only one file number at a time.

The **RS, CS, DS, CD,** and LF options affect the line signals as follows:

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.

The CD (Carrier Detect) is also known as the RLSD (Received Line Signal Detect).

> **Note:** The *speed, parity, data*, and *stop* parameters are positional, but **RS, CS, DS, CD,** and **LF** are not.

The RTS (Request To Send) line is turned on when you execute an OPEN "COM... statement unless you include the **RS** option.

The *n* argument in the **CS, DS,** and **CD** options specifies the number of milliseconds to wait for the signal before returning a "Device Timeout" error. *n* may range from 0 to 65535. If *n* is omitted or is equal to zero, then the line status is not checked at all.

The defaults are **CS1000, DS1000,** and **CD0**. If RS was specified, **CS0** is the default.

That is, normally I/O statements to a communications file will fail if the CTS (Clear To Send) or DSR (Data Set Ready) signals are off. The system waits one second before returning a "Device Timeout." The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the timeout.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM... statement is executed. The **CD** option allows you to test this line by including the *n* parameter, in the same way as **CS** and **DS**. If *n* is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the CD option).

The **LF** parameter is intended for those using communication files as a means of printing to a serial line printer. When you specify **LF**, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). (This includes the carriage return sent as a result of the width setting.) Note that INPUT # and LINE INPUT #, when used to read from a communications file that was opened with the LF option, stop when they see a carriage return. The line feed is always ignored.

Any coding errors within the string expression starting with *speed* results in a "Bad file name" error. An indication as to which parameter is in error is not given.

Refer to "Appendix F. Communications" for more information on control of output signals and other technical information on communications support.

If you specify 8 data bits, you must specify parity N. If you specify 4 data bits, you must specify a parity, that is, N parity is invalid. BASIC uses all 8 bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using PUT), you must specify 8 data bits. (This is not the case if you are sending numeric data *as text*.)

Refer to the previous section for opening devices other than communications devices.

OPEN “COM... Statement

Example:

```
10 OPEN "COM1:" AS 1
```

File 1 is opened for communication with all defaults. The speed is 300 bps with even parity. There will be 7 data bits and one stop bit.

```
10 OPEN "COM1:2400" AS #2
```

File 2 is opened for communication at 2400 bps. Parity, number of data bits, and number of stop bits are defaulted.

```
20 OPEN "COM2:1200,N,8" AS #1
```

File number 1 is opened for asynchronous I/O at 1200 bps, no parity is to be produced or checked, 8-bit bytes will be sent and received, and 1 stop bit will be transmitted.

```
10 OPEN "COM1:9600,N,8,,CS,DS,CD" AS #1
```

Opens COM1: at 9600 bps with no parity and eight data bits. CTS, DSR, and RLSD are not checked.

```
50 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

Opens COM1: at 1200 bps with the defaults of even parity and seven data bits. RTS is sent, CTS is not checked, and “Device Timeout” is given if DSR is not seen within two seconds. Note that the commas are required to indicate the position of the *parity, start*, and *stop* parameters, even though a value is not specified. This is what is meant by *positional* parameters.

An OPEN statement may be used with an ON ERROR statement to make sure a modem is working properly before sending any data. For example, the following program makes sure we get Carrier Detect (CD or RLSD) from the modem before starting. Line 20 is set to timeout after 10 seconds. TRIES is set to 6 so we give up if Carrier Detect is not seen within one minute. Once communication is established, we re-open the file with a shorter delay until timeout.

```
5 TRIES=6
10 ON ERROR GOTO 100
20 OPEN "COM1:300,N,8,2,CS,DS,CD10000" AS #1
30 ON ERROR GOTO 0
40 CLOSE #1 ' works so can continue
50 GOTO 1000
 .
 .
 .
100 TRIES=TRIES-1
110 IF TRIES=0 THEN ON ERROR GOTO 0 ' give up
120 RESUME
 .
 .
 .
1000 OPEN "COM1:300,N,8,2,CS,DS,CD2000" AS #1
```

The next example shows a typical way to use a communication file to control a serial line printer. The LF parameter in the OPEN statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000,DS10000,
     CD10000,LF" AS #1
```

OPTION BASE
Statement

Purpose: Declares the minimum value for array subscripts.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OPTION BASE *n*

Remarks: *n* is 1 or 0.

The default base is 0. If the statement:

```
OPTION BASE 1
```

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

The OPTION BASE statement must be coded *before* you define or use any arrays.

OUT
Statement

Purpose: Sends a byte to a machine output port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: OUT *n,m*

Remarks: *n* is a numeric expression for the port number, in the range 0-65535.

m is a numeric expression for the data to be transmitted, in the range 0-255.

Refer to the *IBM Personal Computer Technical Reference* manual for a description of valid port numbers (I/O addresses).

OUT is the complementary statement to the INP function. Refer to “INP Function” in this chapter.

One use of OUT is to affect the video output. On some displays attached to the Color/Graphics Monitor Adapter, you may find that the first two or three characters on the line don't show up on the screen. If your display does not have a horizontal adjustment control, you can use the following statements to shift the display:

```
OUT 980,2: OUT 981,43
```

This shifts the display two characters to the right in 40-column width (or 16 points in medium resolution graphics mode, or 32 points in high resolution graphics mode).

OUT
Statement

```
OUT 980,2: OUT 981,85
```

This shifts the display right five characters in 80-column width.

The shift caused by these OUT statements remains in effect until a WIDTH or SCREEN statement is executed. The MODE command from DOS can also be used to shift the display as described here; it has the benefit of remaining in effect until a System Reset.

Example:

```
100 OUT 32,100
```

This sends the value 100 to output port 32.

PAINT
Statement

Purpose: Fills in an area on the screen with the selected color.

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

Graphics mode only.

Format: PAINT (*x,y*) [,*paint* [,*boundary*]]

Remarks: (*x,y*) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3). This point will be used as a starting point.

paint is the color to be painted with, in the range 0 to 3. In medium resolution, this color is 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, *paint* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

boundary is the color of the edges of the figure to be filled in, in the range 0 to 3 as described above.

The figure to be filled in is the figure with edges of *boundary* color. The figure is filled in with the color *paint.*

PAINT
Statement

Since there are only two colors in high resolution it doesn't make sense for *paint* to be different from *boundary*. Since *boundary* is defaulted to equal *paint* we don't need the third parameter in high resolution mode.

In high resolution this means "blacking out" an area until black is hit, or "whiting out" an area until white is hit.

In medium resolution we can fill in with color 1 with a border of color 2. Visually this might mean a green ball with a red border.

The starting point of PAINT must be inside the figure to be painted. If the specified point already has the color *boundary* then PAINT will have no effect. If *paint* is omitted the foreground color is used (3 in medium resolution, 1 in high resolution). PAINT can paint any type of figure, but "jagged" edges on a figure will increase the amount of stack space required by PAINT. So if a lot of complex painting is being done you may want to use CLEAR at the beginning of the program to increase the stack space available.

The PAINT statement allows scenes to be displayed with very few statements. This can be a very useful capability.

Example:

```
5 SCREEN 1
10 LINE (0,0)-(100,150),2,B
20 PAINT (50,50),1,2
```

The PAINT statement in line 20 fills in the box drawn in line 10 with color 1.

PEEK
Function

Purpose: Returns the byte read from the indicated memory position.

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

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = PEEK(*n*)

Remarks: *n* is an integer in the range 0 to 65535. *n* is the offset from the current segment as defined by the DEF SEG statement, and indicates the address of the memory location to be read. (See "DEF SEG Statement" in this chapter.)

The returned value will be an integer in the range 0 to 255.

PEEK is the complementary function to the POKE statement (see "POKE Statement," later in this chapter).

Example: The following example can be used in a program to test which display adapter is on the system. After line 30 is executed, the variable IBMMONO will have a value of 0 (zero) if the Color/Graphics Monitor Adapter is used, or 1 (one) if the IBM Monochrome Display and Parallel Printer Adapter is used.

```
10 'test display adapter
20 DEF SEG=0
30 IF (PEEK(&410) AND &H30)=&H30
       THEN IBMMONO=1
       ELSE IBMMONO=0
```

PEN
Statement and Function

Purpose: Reads the light pen.

Versions:

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

PEN STOP only in Advanced and Compiler.

Format: As a statement:

PEN ON

PEN OFF

PEN STOP

As a function:

v = PEN(*n*)

Remarks: The PEN function, *v*=PEN(*n*), reads the light pen coordinates.

n is a numeric expression in the range 0 to 9, and affects the value returned by the function as follows:

0 A flag indicating if pen was down since last poll. Returns -1 if down, 0 if not.

1 Returns the x coordinate where pen was last activated. Range is 0 to 319 in medium resolution, or 0 to 639 in high resolution.

2 Returns the y coordinate where pen was last activated. Range is 0 to 199.

3 Returns the current pen switch value. -1 if down, 0 if up.

4	Returns the last known valid x coordinate. Range is 0 to 319 in medium resolution, or 0 to 639 in high resolution.
5	Returns the last known valid y coordinate. Range is 0 to 199.
6	Returns the character row position where pen was last activated. Range is 1 to 24.
7	Returns the character column position where pen was last activated. Range is 1 to 40 or 1 to 80 depending on WIDTH.
8	Returns the last known valid character row. Range is 1 to 24.
9	Returns the last known valid character column position. Range is 1 to 40 or 1 to 80 depending on WIDTH.

PEN ON enables the PEN read function. The PEN function is initially off. A PEN ON statement must be executed before any pen read function calls can be made. A call to the PEN function while the PEN function is off results in an "Illegal function call" error.

Conversely, for execution speed improvements, it is a good idea to turn the pen off with a PEN OFF statement when you are not using the light pen.

For Advanced BASIC, executing PEN ON will also allow trapping to take place with the ON PEN statement. After PEN ON, if a nonzero line number was specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. Refer to "ON PEN Statement" in this chapter.

PEN
Statement and Function

PEN OFF disables the PEN read function. For Advanced BASIC, no trapping of the pen takes place and action by the light pen is not remembered even if it does take place.

PEN STOP is only available in Advanced BASIC. It disables trapping of light pen activity, but if activity happens it is remembered so an immediate trap occurs when a PEN ON is executed.

When the pen is down in the border area of the screen, the values returned are inaccurate.

You should not attempt I/O to cassette while PEN is ON.

Example:

```
50 PEN ON
60 FOR I=1 TO 500
70 X=PEN(0): X1=PEN(3)
80 PRINT X, X1
90 NEXT
100 PEN OFF
```

This example prints the pen value since the last poll, and the current value.

PLAY
Statement

Purpose: Plays music as specified by *string.*

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: PLAY *string*

Remarks: PLAY implements a concept similar to DRAW by imbedding a "tune definition language" into a character string.

string is a string expression consisting of single character music commands.

The single character commands in PLAY are:

A to G with optional #, +, or –
Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterwards indicates a sharp, a minus sign (-) indicates a flat. The #, +, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n Octave. Sets the current octave for the following notes. There are 7 octaves, numbered 0 to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

N n Plays note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave (O n) and the note name (A–G).

L n Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A.

P n Pause (rest). n may range from 1 to 64, and figures the length of the pause in the same way as L (length).

(dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A.." will play 9/4 as long as L specifies, "A..." will play 27/8 as long, etc. Dots may also appear after a pause (P) to scale the pause length in the same way.

T n Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120. Under "SOUND Statement," later in this chapter, is a table listing common tempos and the equivalent beats per minute.

MF	Music foreground. Music (created by SOUND or PLAY) runs in foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. You can press Ctrl-Break to exit PLAY. Music foreground is the default state.
MB	Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. That is, each note or sound is placed in a buffer allowing the BASIC program to continue executing while music plays in the background. Up to 32 notes (or rests) may be played in background at a time.
MN	Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of MN, ML, and MS.
ML	Music legato. Each note plays the full period set by L (length).
MS	Music staccato. Each note plays 3/4 of the time specified by L.
X variable;	Executes specified string.

In all of these commands the *n* argument can be a constant like 12 or it can be ***=variable;*** where *variable* is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the X command. Otherwise a semicolon is optional between commands, except a semicolon is not allowed after MF, MB, MN, ML, or MS. Also, any blanks in *string* are ignored.

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	**Alternative Method**
`PLAY "XA$;"`	`PLAY "X"+VARPTR$(A$)`
`PLAY "O=I;"`	`PLAY "O="+VARPTR$(I)`

You can use X to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

Example: The following example plays a tune.

```
10 REM little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-."
```

POINT
Function

Purpose: Returns the color of the specified point on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: *v* = POINT (*x,y*)

Remarks: (*x,y*) are the coordinates of the point to be used. The coordinates must be in absolute form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3).

If the point given is out of range the value -1 is returned. In medium resolution valid returns are 0, 1, 2, and 3. In high resolution they are 0 and 1.

Example: The following example inverts the current state of point (I,I).

```
5 SCREEN 2
10 IF POINT(I,I)<>0 THEN PRESET(I,I)
                     ELSE PSET(I,I)
```

or

```
10 PSET(I,I),1-POINT(I,I)
```

POKE
Statement

Purpose: Writes a byte into a memory location.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: POKE *n,m*

Remarks:

n must be in the range 0 to 65535 and indicates the address of the memory location where the data is to be written. It is an offset from the current segment as defined by the DEF SEG statement (see "DEF SEG Statement" in this chapter).

m m is the data to be written to the specified location. It must be in the range 0 to 255.

The complementary function to POKE is PEEK. (See "PEEK Function" in this chapter.) POKE and PEEK are useful for efficient data storage, loading machine language subroutines, and passing arguments and results to and from machine language subroutines.

Warning:
BASIC does not do any checking on the address. So don't go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Example:

```
10 DEF SEG: POKE 106,0
```

See "INKEY$ Variable" in this chapter for an explanation of this example.

POS
Function

Purpose: Returns the current cursor column position.

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

Format: v = POS(n)

Remarks: *n* is a dummy argument.

The current horizontal (column) position of the cursor is returned. The returned value will be in the range 1 to 40 or 1 to 80, depending on the current WIDTH setting. CSRLIN can be used to find the vertical (row) position of the cursor (see "CSRLIN Variable" in this chapter).

Also see the LPOS function.

Example:

```
IF POS(0)>60 THEN PRINT CHR$(13)
```

This example prints a carriage return (moves the cursor to the beginning of the next line) if the cursor is beyond position 60 on the screen.

PRINT Statement

Purpose: Displays data on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT [*list of expressions*] [;]

? [*list of expressions*] [;]

Remarks: *list of expressions*

is a list of numeric and/or string expressions, separated by commas, blanks, or semicolons. Any string constants in the list must be enclosed in quotation marks.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

> **Note:** The question mark (?) may be used as a shorthand way of entering PRINT only when you are using the BASIC program editor.

Print Positions

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

If a comma, semicolon, or SPC or TAB function ends the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions ends without a comma, semicolon, SPC or TAB function, a carriage return is printed at the end of the line; that is, BASIC moves the cursor to the beginning of the next line.

If the length of the value to be printed exceeds the number of character positions remaining on the current line, then the value will be printed at the beginning of the next line. If the value to be printed is longer than the defined WIDTH, BASIC prints as much as it can on the current line and continues printing the rest of the value on the next physical line.

Scrolling occurs as described under "Text Mode" in Chapter 3.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers that can be represented with 7 or fewer digits in fixed point format no less accurately than they can be represented in the floating point format, are output using fixed point or integer format. For example, 10^(-7) is output as .0000001 and 10^(-8) is output as 1E-8.

BASIC automatically inserts a carriage return/line feed after printing *width* characters, where *width* is 40 or 80, as defined by the WIDTH statement. This will cause two lines to be skipped when you print exactly 40 (or 80) characters, unless the PRINT statement ends in a semicolon (;).

LPRINT is used to print information on the printer. See "LPRINT and LPRINT USING Statements" earlier in this chapter.

PRINT Statement

Example:

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

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

```
Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729
Ok
RUN
? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
Ok
```

Here, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line.

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

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

PRINT USING
Statement

Purpose: Prints strings or numbers using a specified format.

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

Format: PRINT USING *v$; list of expressions* [;]

Remarks: *v$* is a string constant or variable which consists of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

list of expressions
consists of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas.

String Fields

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

!	Specifies that only the first character in the given string is to be printed.
\n spaces\	Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and so on.

If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

Example:

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

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

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

Numeric Fields

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

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

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

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

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

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

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

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

– A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

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

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

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks.

The ** also specifies positions for two more digits.

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

$$

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

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

**$

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

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

A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position.

The comma has no effect if used with the exponential (^^^^) format.

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

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

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

```
Ok
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok
PRINT USING ".###^^^^-";-88888
.889E+05-
Ok
PRINT USING "+.##^^^^";123
+.12E+03
Ok
```

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

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

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

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

```
Ok
PRINT USING "##.##";111.22
%111.22
Ok
PRINT USING ".##";.999
%1.00
Ok
```

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

Example: This example shows how you can include string constants in the format string.

```
Ok
PRINT USING "THIS IS EXAMPLE _##"; 1
THIS IS EXAMPLE #1
Ok
```

PRINT # and PRINT # USING Statements

Purpose: Writes data sequentially to a file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT *#filenum*, [USING *v$*;] *list of exps*

Remarks:

filenum	is the number used when the file was opened for output.
v$	is a string expression comprised of formatting characters as described in the previous section, "PRINT USING Statement."
list of exps	is a list of the numeric and/or string expressions that will be written to the file.

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

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

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

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

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

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

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

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

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

The image written to the file is

```
CAMERA,93604-1
```

which can be read back into two string variables.

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

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

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

writes the following image to the file:

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

and the statement:

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

inputs the string “CAMERA” to A$ and “AUTOMATIC   93604-1” to B$.

To separate these strings properly on the file, write double quotes to the file image using CHR$(34). The statement:

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

writes the following image to the file:

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

and the statement:

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

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

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

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

The easy way to avoid all these problems is to use the WRITE # statement rather than the PRINT # statement. (Refer to "WRITE # Statement," at the end of this chapter.)

Example: For more examples using PRINT # and WRITE #, see "Appendix B. BASIC Diskette Input and Output."

PSET and PRESET Statements

Purpose: Draws a point at the specified position on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Graphics mode only.

Format: PSET (*x,y*) [,*color*]

PRESET (*x,y*) [,*color*]

Remarks:

(*x,y*)	are the coordinates of the point to be set. They may be in absolute or relative form, as explained in the section "Specifying Coordinates" under "Graphics Modes" in Chapter 3.
color	specifies the color to be used, 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. In high resolution a color value of 2 will be treated as 0, and 3 will be treated as 1.

PRESET is almost identical to PSET. The only difference is that if no *color* parameter is given to PRESET, the background color (0) is selected. If *color* is included, PRESET is identical to PSET. Line 70 in the example below could just as easily be:

```
70 PSET(I,I),0
```

If an out of range coordinate is given to PSET or PRESET no action is taken nor is an error given. If *color* is greater than 3, this will result in an "Illegal function call" error.

Example: Lines 20 through 40 of this example draw a diagonal line from the point (0,0) to the point (100,100). Then lines 60 through 80 erase the line by setting each point to a color of 0.

```
10 SCREEN 1
20 FOR I=0 TO 100
30 PSET (I,I)
40 NEXT
50 'erase line
60 FOR I=100 TO 0 STEP -1
70 PRESET(I,I)
80 NEXT
```

PUT
Statement (Files)

Purpose: Writes a record from a random buffer to a random file.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: PUT [#] *filenum* [,*number*]

Remarks:

filenum	is the number under which the file was opened.
number	is the record number for the record to be written, in the range 1 to 32767.

If *number* is omitted, the record has the next available record number (after the last PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET may be used to put characters in the random file buffer before a PUT statement. In the case of WRITE #, BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a "Field overflow" error. Refer to "Appendix B. BASIC Diskette Input and Output."

Because BASIC and DOS block as many records as possible in 512 byte sectors, the PUT statement does not necessarily perform a physical write to the diskette.

PUT
Statement (Files)

PUT can be used for a communications file. In that case *number* is the number of bytes to write to the communications file. This number must be less than or equal to the value set by the LEN option on the OPEN "COM... statement.

Example: See "Appendix B. BASIC Diskette Input and Output."

PUT
Statement (Graphics)

Purpose: Writes colors onto a specified area of the screen.

Versions:

Cassette	Disk	Advanced	Compiler
		***	***

Graphics mode only.

Format: PUT (*x,y*) *,array* [*,action*]

Remarks:

(*x,y*)	are the coordinates of the top left corner of the image to be transferred.
array	is the name of a numeric array containing the information to be transferred. See "GET Statement (Graphics)" in this chapter for more information on this array.
action	is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it onto the screen. However it also provides the option of interacting with the data already on the screen by the use of the action.

PSET as an action simply stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except a negative image is produced. That is, a value of 0 in the array causes the corresponding point to have color number 3, and vice versa; a value of 1 in the array causes the corresponding point to have color number 2, and vice versa.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode which may be used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background *twice*, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution mode, AND, XOR, and OR have the following effects on color:

AND

		array value			
		0	1	2	3
screen	0	0	0	0	0
	1	0	1	0	1
	2	0	0	2	2
	3	0	1	2	3

PUT
Statement (Graphics)

OR

		array value			
		0	1	2	3
screen	0	0	1	2	3
	1	1	1	3	3
	2	2	3	2	3
	3	3	3	3	3

XOR

		array value			
		0	1	2	3
screen	0	0	1	2	3
	1	1	0	3	2
	2	2	3	0	1
	3	3	2	1	0

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR).
2. Recalculate the new position of the object.
3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image.
4. Go to step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after" images of the object. This way the extra area will effectively erase the old image. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an "Illegal function call" error occurs.

RANDOMIZE
Statement

Purpose: Reseeds the random number generator.

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

Format: RANDOMIZE [*n*]

Remarks: *n* is an integer expression which will be used as the random number seed.

If *n* is omitted, BASIC suspends program execution and asks for a value by displaying:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

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

In Disk and Advanced BASIC, the internal clock can be a useful way to get a random number seed. You can use VAL to change the last two digits of TIME$ to a number, and use that number for *n*.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose you respond with 3. The program continues:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose this time you respond with 4. The program continues:

```
Random Number Seed (-32768 to 32767)? 4
 .1719568  .5273236  .6879686  .713297
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

If you try 3 again, you'll get the same sequence as the first run:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
```

READ
Statement

Purpose: Reads values from a DATA statement and assigns them to variables (see "DATA Statement" in this chapter).

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

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

Remarks: *variable* is a numeric or string variable or array element which is to receive the value read from the DATA table.

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

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

READ
Statement

To reread data from any line in the list of DATA statements, use the RESTORE statement (see "RESTORE Statement" in this chapter).

Example:

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

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

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

This program reads string and numeric data from the DATA statement in line 30. Note that you don't need quotation marks around COLORADO, because it doesn't have commas, semicolons, or significant leading or trailing blanks. However, you do need the quotation marks around "DENVER," because of the comma.

REM Statement

Purpose: Inserts explanatory remarks in a program.

Versions:

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

Format: REM *remark*

Remarks: *remark* may be any sequence of characters.

REM statements are not executed but are output exactly as entered when the program is listed. However, they do slow up execution time somewhat, and take up space in memory.

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

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM. If you put a remark on a line with other BASIC statements, the remark must be the *last* statement on the line.

Example:

```
100 REM calculate average velocity
110 SUM=0: REM initialize SUM
120 FOR I=1 TO 20
130 SUM=SUM + V(I)
  .
  .
  .
```

Line 110 might also be written:

```
110 SUM=0 ' initialize SUM
```

RENUM
Command

Purpose: Renumbers program lines.

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

Format: RENUM [*newnum*] [,[*oldnum*] [,*increment*]]

Remarks: *newnum* is the first line number to be used in the new sequence. The default is 10.

oldnum is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the increment to be used in the new sequence. The default is 10.

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

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

RENUM Command

Example: `RENUM`

Renumbers the entire program. The first new line number is 10. Lines increment by 10.

```
RENUM 300,,50
```

Renumbers the entire program. The first new line number is 300. Lines increment by 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET
Command

Purpose:	Closes all diskette files and clears the system buffer.			
Versions:	Cassette	Disk ***	Advanced ***	Compiler ***
Format:	RESET			
Remarks:	If all open files are on diskette, then RESET is the same as CLOSE with no file numbers after it.			

RESTORE
Statement

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

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

Format: RESTORE [*line*]

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

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
Ok
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A;B;C;D;E;F
RUN
 57  68  79  57  68  79
Ok
```

The RESTORE statement in line 20 resets the DATA pointer to the beginning, so that the values that are read in line 30 are 57, 68, and 79.

RESUME
Statement

Purpose: Continues program execution after an error recovery procedure is performed.

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

Format: RESUME [0]

RESUME NEXT

RESUME *line*

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

RESUME or RESUME 0

Execution resumes at the statement which caused the error.

> **Note:** If you try to renumber a program containing a RESUME 0 statement, you will get an "Undefined line number" error. The statement will still say RESUME 0, which is okay.

RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME line	Execution resumes at the specified line number.

RESUME
Statement

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

Example:

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

Line 900 is the beginning of the error trapping routine. The RESUME statement causes the program to return to line 80 when error 230 occurs in line 90.

RETURN
Statement

Purpose: To bring you back from a subroutine. See "GOSUB and RETURN Statements" in this chapter.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

line valid only in Advanced and Compiler.

Format: RETURN [*line*]

Remarks: *line* is the line number of the program line you wish to return to. You may use it only in Advanced BASIC.

Although you can use RETURN *line* to return from any subroutine, this enhancement was added to allow non-local returns from the event trapping routines. From one of these routines you will often want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. Use of the non-local RETURN must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

RIGHT$ Function

Purpose: Returns the rightmost *n* characters of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = RIGHT$(*x$*,*n*)

Remarks: *x$* is any string expression.

n is an integer expression specifying the number of characters to be in the result.

If *n* is greater than or equal to LEN(*x$*), then *x$* is returned. If *n* is zero, the null string (length zero) is returned.

Also see the MID$ and LEFT$ functions.

Example:

```
Ok
10 A$="BOCA RATON, FLORIDA"
20 PRINT RIGHT$(A$,7)
RUN
FLORIDA
Ok
```

The rightmost seven characters of the string A$ are returned.

RND
Function

Purpose: Returns a random number between 0 and 1.

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

Format: *v* = RND[(*x*)]

Remarks: *x* is a numeric expression which affects the returned value as described below.

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. This is most easily done using the RANDOMIZE statement (see “RANDOMIZE Statement” in this chapter). You may also reseed the generator when you call the RND function by using *x* where *x* is negative. This always generates the particular sequence for the given *x*. This sequence is not affected by RANDOMIZE, so if you want to generate a different sequence each time the program is run, you must use a different value for *x* each time.

If *x* is positive or not included, RND(*x*) generates the next random number in the sequence.

RND(0) repeats the last number generated.

To get random numbers in the range 0 (zero) through *n*, use the formula:

```
INT( RND * (n+1))
```

RND
Function

Example:

```
Ok
10 FOR I=1 TO 3
20 PRINT RND(I);     ' x>0
30 NEXT I
40 PRINT: X=RND(-6) ' x<0
50 FOR I=1 TO 3
60 PRINT RND(I);     ' x>0
70 NEXT I
80 RANDOMIZE 853 'randomize
90 PRINT: X=RND(-6) ' x<0
100 FOR I=1 TO 3
110 PRINT RND;    ' same as x>0
120 NEXT I
130 PRINT: PRINT RND(0)
RUN
 .6291626  .1948297  .6305799
 .6818615  .4193624  .6215937
 .6818615  .4193624  .6215937
 .6215937
Ok
```

The first horizontal line of results shows three random numbers, generated using a positive *x*.

In line 40, a negative number is used to reseed the random number generator. The random numbers produced after this seeding are in the second row of results.

In line 80, the random number generator is reseeded using the RANDOMIZE statement; in line 90 it is reseeded again by calling RND with the same negative value we used in line 40. This cancels the effect of the RANDOMIZE statement, as you can see; the third line of results is identical to the second line.

In line 130, RND is called with an argument of zero, so the last number printed is the same as the preceding number.

RUN
Command

Purpose: Begins execution of a program.

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

Format: RUN [*line*]

RUN *filespec*[,R]

Remarks: *line* is the line number of the program in memory where you wish execution to begin.

filespec is a string expression for the file specification, as explained under "Naming Files" in Chapter 3. The default extension .BAS is supplied for diskette files.

RUN or RUN *line* begins execution of the program currently in memory. If *line* is specified, execution begins with the specified line number. Otherwise, execution begins at the lowest line number.

RUN *filespec* loads a file from diskette or cassette into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain open. Refer also to "Appendix B. BASIC Diskette Input and Output."

Executing a RUN command will turn off any sound that is running and reset to Music Foreground. Also, PEN and STRIG will be reset to OFF.

RUN
Command

Example:

```
Ok
10 PRINT 1/7
RUN
 .1428571
Ok
10 PI=3.141593
20 PRINT PI
RUN 20
 0
Ok
```

In this first example, we use the first form of RUN on two very small programs. The first program is run from the beginning. We used the RUN *line* option for the second example to run the program from line 20. In this case, line 10 does not get executed, so PI does not receive its proper value. A 0 is printed because all numeric variables have an initial value of zero.

```
RUN "CAS1:NEWFIL",R
```

The preceding example loads the program "NEWFIL" from the tape and runs it, keeping files open.

SAVE
Command

Purpose: Saves a BASIC program file on diskette or cassette.

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

Format: SAVE *filespec* [,A]

SAVE *filespec* [,P]

Remarks: *filespec* is a string expression for the file specification. If *filespec* does not conform to the rules outlined under "Naming Files" in Chapter 3, an error is issued and the save is cancelled.

The BASIC program is written to the specified device. When saving to CAS1:, the cassette motor is turned on and the file is immediately written to the tape.

For diskette files, if the filename is eight characters or less and no extension is supplied, the extension .BAS is added to the name. If a file with the same filename already exists on the diskette, it will be written over.

When using Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for SAVE in Cassette BASIC.

For Disk and Advanced BASIC, the device defaults to the DOS default drive.

The A option saves the program in ASCII format. Otherwise, BASIC saves the file in a compressed binary (tokenized) format. ASCII files take up more space, but some types of access require that files be in ASCII format. For example, a file intended to be merged must be saved in ASCII format. Programs saved in ASCII may be read as data files.

The **P** option saves the program in an encoded binary format. This is the protection option. When a protected program is later run (or loaded), any attempt to LIST or EDIT it fails with an "Illegal function call" error. No way is provided to "unprotect" such a program.

> **Note:** The diskette directory entry for a BASIC program file gives no indication that the file is either protected or stored in ASCII format. The **.BAS** extension is used in any case.

See also "Appendix B. BASIC Diskette Input and Output."

Example:

```
SAVE "INVENT"
```

Saves the program in memory as INVENT. The program is saved on cassette if you are using Cassette BASIC. If you are using Disk or Advanced BASIC, the program is saved on the diskette in the DOS default drive and given an extension of **.BAS**.

```
SAVE "B:PROG",A
```

Saves PROG.BAS on drive B: in ASCII, so it may later be merged.

```
SAVE "A:SECRET.BOZ",P
```

Saves SECRET.BOZ on drive A:, protected so it may not be altered.

SCREEN
Function

Purpose: Returns the ASCII code (0-255) for the character on the active screen at the specified row (line) and column.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = SCREEN(*row,col*[,*z*])

Remarks: *row* is a numeric expression in the range 1 to 25.

col is a numeric expression in the range 1 to 40 or 1 to 80 depending upon the WIDTH setting.

z is a numeric expression which evaluates to a true or false value. *z* is only valid in text mode.

Refer to "Appendix G. ASCII Character Codes" for a list of ASCII codes.

In text mode, if *z* is included and is true (non-zero), the color attribute for the character is returned instead of the code for the character. The color attribute is a number in the range 0 to 255. This number, *v*, may be deciphered as follows:

(*v* MOD 16) is the foreground color.

(((*v* - *foreground*)/16) MOD 128) is the background color, where *foreground* is calculated as above.

(*v*>127) is true (-1) if the character is blinking, false (0) if not.

SCREEN Function

Refer to "COLOR Statement" for a list of colors and their associated numbers.

In graphics mode, if the specified location contains graphic information (points or lines, as opposed to just a character), then the SCREEN function returns zero.

Any values entered outside of the ranges indicated result in an "Illegal function call" error.

The SCREEN *statement* is explained in the next section.

Example:

```
100 X = SCREEN (10,10)
```

If the character at 10,10 is A, then X is 65.

```
110 X = SCREEN (1,1,1)
```

Returns the color attribute of the character in the upper left hand corner of the screen.

SCREEN
Statement

Purpose: Sets the screen attributes to be used by subsequent statements.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Meaningful with the Color/Graphics Monitor Adapter only.

Format: SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]]]

Remarks: *mode* is a numeric expression resulting in an integer value of 0, 1 or 2. Valid modes are:

0 Text mode at current width (40 or 80).

1 Medium resolution graphics mode (320x200). Use with Color/Graphics Monitor Adapter only.

2 High resolution graphics mode (640x200). Use with Color/Graphics Monitor Adapter only.

burst is a numeric expression resulting in a true or false value. It enables color. In text mode (*mode*=0), a false (zero) value disables color (black and white images only) and a true (non-zero) value enables color (allows color images). In medium resolution graphics mode (*mode*=1), a true (non-zero) value will disable color, and a false (zero) value will enable color. Since black and white are the only colors in high resolution graphics (*mode*=2), this parameter will not have much effect in high resolution.

apage	(active page) is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the page to be written to by output statements to the screen, and is valid in text mode (*mode*=0) only.
vpage	(visual page) selects which page is to be displayed on the screen, in the same way as *apage* above. The visual page may be different than the active page. *vpage* is valid in text mode (*mode*=0) only. If omitted, *vpage* defaults to *apage*.

If all parameters are valid, the new screen mode is stored, the screen is erased, the foreground color is set to white, and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is text, and only *apage* and *vpage* are specified, the effect is that of changing display pages for viewing. Initially, both active and visual pages default to 0 (zero). By manipulating active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

> **Note:** There is only one cursor shared between all the pages. If you are going to switch active pages back and forth, you should save the cursor position on the current active page (using POS(0) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

Any parameter may be omitted. Omitted parameters, except *vpage*, assume the old value.

Any values entered outside of the ranges indicated will result in an "Illegal function call" error. Previous values are retained.

If you are writing a program which is intended to be run on a machine that may have either adapter, we suggest you use the SCREEN 0,0,0 and WIDTH 40 statements at the beginning of the program.

Example:

```
10 SCREEN 0,1,0,0
```

Selects text mode with color, and sets active and visual page to 0.

```
20 SCREEN ,,1,2
```

Mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

```
30 SCREEN 2,,0,0
```

Switches to high resolution graphics mode.

```
40 SCREEN 1,0
```

Switches to medium resolution color graphics.

```
50 SCREEN ,1
```

Sets medium resolution graphics with color off.

SGN
Function

Purpose: Returns the sign of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = SGN(*x*)

Remarks: *x* is any numeric expression.

SGN(*x*) is the mathematical signum function:

- If *x* is positive, SGN(*x*) returns 1.
- If *x* is zero, SGN(*x*) returns 0.
- If *x* is negative, SGN(*x*) returns -1.

Example:

```
ON SGN(X)+2 GOTO 100,200,300
```

branches to 100 if X is negative, 200 if X is zero, and 300 if X is positive.

SIN
Function

Purpose: Calculates the trigonometric sine function.

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

Format: v = SIN(x)

Remarks: x is an angle in radians.

If you want to convert degrees to radians, multiply by PI/180, where PI=3.141593.

SIN(x) is calculated in single precision.

Example:

```
Ok
10 PI=3.141593
20 DEGREES = 90
30 RADIANS=DEGREES * PI/180 ' PI/2
40 PRINT SIN(RADIANS)
RUN
 1
Ok
```

This example calculates the sine of 90 degrees, after first converting the degrees to radians.

SOUND
Statement

Purpose: Generates sound through the speaker.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SOUND *freq, duration*

Remarks:

freq	is the desired frequency in Hertz (cycles per second). It must be a numeric expression in the range 37 to 32767.
duration	is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. *duration* must be a numeric expression in the range 0 to 65535.

When the SOUND statement produces a sound, the program continues to execute until another SOUND statement is reached. If *duration* of the new SOUND statement is zero, the current SOUND statement that is running is turned off. Otherwise, the program waits until the first sound completes before it executes the new SOUND statement.

If you are using Advanced BASIC, you can cause the sounds to be buffered so execution does not stop when a new SOUND statement is encountered. See the MB command explained under "PLAY Statement" in this chapter for details.

If no SOUND statement is running, SOUND *x*,0 has no effect.

The tuning note, A, has a frequency of 440. The following table correlates notes with their frequencies for two octaves on either side of middle C.

Note	Frequency	Note	Frequency
C	130.810	C*	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

*middle C. Higher (or lower) notes may be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,*duration*.

The duration for one beat can be calculated from beats per minute by dividing the beats per minute into 1092 (the number of clock ticks in a minute).

The next table shows typical tempos in terms of clock ticks:

	Tempo	Beats/ Minute	Ticks/ Beat
very slow	Larghissimo		
	Largo	40-60	27.3-18.2
	Larghetto	60-66	18.2-16.55
	Grave		
	Lento		
↓	Adagio	66-76	16.55-14.37
slow	Adagietto		
↓	Andante	76-108	14.37-10.11
medium	Andantino		
↓	Moderato	108-120	10.11-9.1
fast	Allegretto		
	Allegro	120-168	9.1-6.5
	Vivace		
	Veloce		
↓	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Example: The following program creates a glissando up and down.

```
10 FOR I=440 TO 1000 STEP 5
20 SOUND I, 0.5
30 NEXT
40 FOR I=1000 TO 440 STEP -5
50 SOUND I, 0.5
60 NEXT
```

SPACE$
Function

Purpose: Returns a string consisting of *n* spaces.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = SPACE$(*n*)

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

Refer also to the SPC function.

Example:

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

This example uses the SPACE$ function to print each number I on a line preceded by I spaces. An additional space is inserted because BASIC puts a space in front of positive numbers.

SPC
Function

Purpose: Skips *n* spaces in a PRINT statement.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT SPC(*n*)

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

If *n* is greater than the defined width of the device, then the value used is *n* MOD *width*. SPC may only be used with PRINT, LPRINT and PRINT # statements.

If the SPC function is at the end of the list of data items, then BASIC does not add a carriage return, as though the SPC function had an implied semicolon after it.

Also see the SPACE$ function.

Example:

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

This example prints OVER and THERE separated by 15 spaces.

SQR
Function

Purpose: Returns the square root of *x*.

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

Format: $v = \mathrm{SQR}(x)$

Remarks: *x* must be greater than or equal to zero.

Example

```
Ok
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.162278
 15            3.872984
 20            4.472136
 25            5
Ok
```

This example calculates the square roots of the numbers 10, 15, 20 and 25.

STICK
Function

Purpose: Returns the x and y coordinates of two joysticks.

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

Format: *v* = STICK(*n*)

Remarks: *n* is a numeric expression in the range 0 to 3 which affects the result as follows:

0 returns the x coordinate for joystick A.

1 returns the y coordinate of joystick A.

2 returns the x coordinate of joystick B.

3 returns the y coordinate of joystick B.

Note: STICK(0) retrieves all four values for the coordinates, and returns the value for STICK(0). STICK(1), STICK(2), and STICK(3) do not sample the joystick. They get the values previously retrieved by STICK (0).

The range of values for x and y depends on your particular joysticks.

STICK
Function

Example:

```
10 PRINT "Joystick B"
20 PRINT "x coordinate","y coordinate"
30 FOR J=1 TO 100
40 TEMP=STICK(0)
50 X=STICK(2): Y=STICK(3)
60 PRINT X,Y
70 NEXT
```

This program takes 100 samples of the coordinates of joystick B and prints them.

STOP
Statement

Purpose: Terminates program execution and returns to command level.

Versions:

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

Format: STOP

Remarks: STOP statements may be used anywhere in a program to terminate execution. When BASIC encounters a STOP statement, it displays the following message:

```
Break in nnnnn
```

where nnnnn is the line number where the STOP occurred.

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after it executes a STOP. You can resume execution of the program by issuing a CONT command (see “CONT Command” in this chapter).

STOP
Statement

Example:

```
10 INPUT A, B
20 TEMP= A*B
30 STOP
40 FINAL = TEMP+200: PRINT FINAL
RUN
? 26, 2.1
Break in 30
Ok
PRINT TEMP
 54.6
Ok
CONT
 254.6
Ok
```

This example calculates the value of TEMP, then stops. While the program is stopped, we can check the value of TEMP. Then we can use CONT to resume program execution at line 40.

STR$ Function

Purpose: Returns a string representation of the value of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STR$(*x*)

Remarks: *x* is any numeric expression.

If *x* is positive, the string returned by STR$ contains a leading blank (the space reserved for the plus sign). For example:

```
Ok
? STR$(321); LEN(STR$(321))
 321 4
Ok
```

The VAL function is complementary to STR$.

Example: This example branches to different sections of the program based on the number of digits in a number that is entered. The digits in the number are counted by using STR$ to convert the number to a string, then branching based on the length of the string.

```
5 REM arithmetic for kids
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300
 •
 •
 •
```

STRIG
Statement and Function

Purpose: Returns the status of the joystick buttons (triggers).

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: As a statement:

STRIG ON

STRIG OFF

As a function:

v = STRIG(*n*)

Remarks: *n* is a numeric expression in the range 0 to 3. It affects the value returned by the function as follows:

0	Returns -1 if button A1 was pressed since the last STRIG(0) function call, returns 0 if not.
1	Returns -1 if button A1 is currently pressed, returns 0 if not.
2	Returns -1 if button B1 was pressed since the last STRIG(2) function call, returns 0 if not.
3	Returns -1 if button B1 is currently pressed, returns 0 if not.

STRIG
Statement and Function

In Advanced BASIC and the BASIC Compiler, you can read four buttons from the joysticks. The additional values for *n* are:

4 Returns -1 if button A2 was pressed since the last STRIG(4) function call, returns 0 if not.

5 Returns -1 if button A2 is currently pressed, returns 0 if not.

6 Returns -1 if button B2 was pressed since the last STRIG(6) function call, returns 0 if not.

7 Returns -1 if button B2 is currently pressed, returns 0 if not.

STRIG ON must be executed before any STRIG(*n*) function calls may be made. After STRIG ON, every time the program starts a new statement BASIC checks to see if a button has been pressed.

If STRIG is OFF, no testing takes place.

Refer also to the next section, "STRIG(n) Statement" for enhancements to the STRIG function in Advanced BASIC.

STRIG(n) Statement

Purpose: Enables and disables trapping of the joystick buttons.

Versions:

Cassette	Disk	Advanced	Compiler
		***	(**)

Format: STRIG(*n*) ON

STRIG(*n*) OFF

STRIG(*n*) STOP

Remarks: *n* may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

STRIG(*n*) ON must be executed to enable trapping by the ON STRIG(*n*) statement (see "ON STRIG(n) Statement" in this chapter). After STRIG(*n*) ON, every time the program starts a new statement, BASIC checks to see if the specified button has been pressed.

If STRIG(*n*) OFF is executed, no testing or trapping takes place. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place. However, if the button is pressed it is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

Refer also to the previous section, "STRIG Statement and Function."

STRING$ Function

Purpose: Returns a string of length *n* whose characters all have ASCII code *m* or the first character of *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v$* = STRING$(*n,m*)

*v$* = STRING$(*n,x$*)

Remarks: *n, m* are in the range 0 to 255.

x$ is any string expression.

Example:

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

The first example repeats an ASCII value of 45 to print a string of hyphens.

```
Ok
10 X$=''ABCD''
20 Y$+STRING$(10,X$)
30 PRINT Y$
RUN
AAAAAAAAAA
Ok
```

The second example repeats the first character of the string "ABCD".

SWAP
Statement

Purpose: Exchanges the values of two variables.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: SWAP *variable1, variable2*

Remarks: *variable1, variable2*

are the names of two variables or array elements.

Any type variable may be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
Ok
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

After line 30 is executed, A$ has the value " ALL " and B$ has the value " ONE ".

SYSTEM Command

Purpose: Exits BASIC and returns to DOS.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: SYSTEM

Remarks: SYSTEM closes all files before it returns to DOS. Your BASIC program is lost.

If you entered BASIC through a Batch file from DOS, the SYSTEM command returns you to the Batch file, which continues executing at the point it left off.

TAB Function

Purpose: Tabs to position *n*.

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

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: PRINT TAB(*n*)

Remarks: *n* must be in the range 1 to 255.

If the current print position is already beyond space *n*, TAB goes to position *n* on the next line. Space 1 is the leftmost position, and the rightmost position is the defined WIDTH.

TAB may only be used in PRINT, LPRINT, and PRINT # statements.

If the TAB function is at the end of the list of data items, then BASIC does not add a carriage return, as though the TAB function had an implied semicolon after it.

Example: TAB is used in the following example to cause the information on the screen to line up in columns.

```
Ok
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "L. M. JACOBS","$25.00"
RUN
NAME                    AMOUNT

L. M. JACOBS            $25.00
Ok
```

TAN Function

Purpose: Returns the trigonometric tangent of *x*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = TAN(*x*)

Remarks:

x	is the angle in radians. To convert degrees to radians, multiply by PI/180, where PI=3.141593.
TAN(*x*)	is calculated in single precision.

Example:

```
Ok
10 PI=3.141593
20 DEGREES=45
30 PRINT TAN(DEGREES*PI/180)
RUN
 1
Ok
```

This example calculates the tangent of 45 degrees.

TIME$
Variable and Statement

Purpose: Sets or retrieves the current time.

Versions:

Cassette	Disk	Advanced	Compiler
	***	***	***

Format: As a variable:

*v$* = TIME$

As a statement:

TIME$ = *x$*

Remarks: **For the variable (v$ = TIME$):**

The current time is returned as an 8 character string. The string is of the form *hh:mm:ss*, where *hh* is the hour (00 to 23), *mm* is the minutes (00 to 59), and *ss* is the seconds (00 to 59). The time may have been set by DOS prior to entering BASIC.

For the statement (TIME$ = x$):

The current time is set. *x$* is a string expression indicating the time to be set. *x$* may be given in one of the following forms:

hh	Set the hour in the range 0 to 23. Minutes and seconds default to 00.
hh:mm	Set the hour and minutes. Minutes must be in the range 0 to 59. Seconds default to 00.
hh:mm:ss	Set the hour, minutes, and seconds. Seconds must be in the range 0 to 59.

TIME$
Variable and Statement

A leading zero may be omitted from any of the above values, but you must include at least one digit. For example, if you wanted to set the time as a half hour after midnight, you could enter TIME$="0:30", but not TIME$=":30". If any of the values are out of range, an "Illegal function call" error is issued. The previous time is retained. If *x$* is not a valid string, a "Type mismatch" error results.

Example: The following program displays the time continuously in the middle of the screen.

```
10 CLS
20 LOCATE 10,15
30 PRINT TIME$
40 GOTO 30
```

TRON and TROFF Commands

Purpose: Traces the execution of program statements.

Versions:

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

Format: TRON

TROFF

Remarks: As an aid in debugging, the TRON command (which may be entered in indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace is turned off by the TROFF command.

Example:

```
Ok
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
TRON
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

This example uses TRON and TROFF to trace execution of a loop. The numbers in brackets are line numbers; the numbers not in brackets at the end of each line are the values of J, K, and L which are printed by the program.

USR Function

Purpose: Calls the indicated machine language subroutine with the argument *arg*.

Versions:

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

Format: *v* = USR[*n*](*arg*)

Remarks:

n	is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for the desired routine (see "DEF USR Statement" in this chapter). If *n* is omitted, USR0 is assumed.
arg	is any numeric expression or string variable, which will be the argument to the machine language subroutine.

The CALL statement is another way to call a machine language subroutine. See "Appendix C. Machine Language Subroutines" for complete information on using machine language subroutines.

Example:

```
10 DEF USR0 = &HF000
50 C = USR0(B/2)
60 D = USR(B/3)
```

The function USR0 is defined in line 10. Line 50 calls the function USR0 with the argument B/2. Line 60 calls USR0 again, with the argument B/3.

VAL
Function

Purpose: Returns the numerical value of string *x$*.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VAL(*x$*)

Remarks: *x$* is a string expression.

The VAL function strips blanks, tabs, and line feeds from the argument string in order to determine the result. For example,

VAL("   -3")

returns -3.

If the first characters of *x$* are not numeric, then VAL(*x$*) will return 0 (zero).

See the STR$ function for numeric to string conversion.

Example:

```
Ok
PRINT VAL("3408 SHERWOOD BLVD.")
 3408
Ok
```

In this example, VAL is used to extract the house number from an address.

VARPTR Function

Purpose: Returns the address in memory of the variable or file control block.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: *v* = VARPTR(*variable*)

v = VARPTR(#*filenum*)

Remarks:

variable	is the name of a numeric or string variable or array element in your program. A value must be assigned to *variable* prior to the call to VARPTR, or an "Illegal function call" error results.
filenum	is the number under which the file was opened.

For both formats, the address returned is an integer in the range 0 to 65535. This number is the offset into BASIC's Data Segment. The address is not affected by the DEF SEG statement.

The first format returns the address of the first byte of data identified with *variable*. The format of this data is described in Appendix I under "How Variables Are Stored."

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

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

The second format returns the starting address of the file control block for the specified file. This is not the same as the DOS file control block. Refer to "BASIC File Control Block" in "Appendix I. Technical Information and Tips" for detailed information about the format of the file control block.

VARPTR is meaningless for cassette files.

Example: This example reads the first byte in the buffer of a random file:

```
10 OPEN "DATA.FIL" AS #1
20 GET #1
30 'get address of control block
40 FCBADR = VARPTR(#1)
50 'figure address of data buffer
60 DATADR = FCBADR+188
70 'get first byte in data buffer
80 A% = PEEK(DATADR)
```

The next example use VARPTR to get the data from a variable. In line 30, P gets the address of the data. Integer data is stored in two bytes, with the less significant byte first. The actual value stored at location P is calculated in line 40. The bytes are read with the PEEK function, and the second byte is multiplied by 256 because it contains the high-order bits.

```
10 DEFINT A-Z
20 DATA1=500
30 P=VARPTR(DATA1)
40 V=PEEK(P) + 256*PEEK(P+1)
50 PRINT V
```

VARPTR$ Function

Purpose: Returns a character form of the address of a variable in memory. It is primarily for use with PLAY and DRAW in programs that will later be compiled.

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

Format: *v$* = VARPTR$(*variable*)

Remarks: *variable* is the name of a variable existing in the program.

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

VARPTR$ returns a three-byte string in the form:

Byte 0	**Byte 1**	**Byte 2**
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2	integer
3	string
4	single-precision
8	double-precision

VARPTR$ Function

The returned value is the same as:

CHR$(*type*)+MKI$(VARPTR(*variable*))

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Method One	**Alternative Method**
`PLAY "XA$;"`	`PLAY "X"+VARPTR$(A$)`
`PLAY "O=I;"`	`PLAY "O="+VARPTR$(I)`

This technique is mainly for use in programs which will later be compiled.

WAIT
Statement

Purpose: Suspends program execution while monitoring the status of a machine input port.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WAIT *port, n*[,*m*]

Remarks:

port	is the port number, in the range 0 to 65535.
n, m	are integer expressions in the range 0 to 255.

Refer to the *IBM Personal Computer Technical Reference* manual for a description of valid port numbers (I/O addresses).

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *m* is omitted, it is assumed to be zero.

WAIT
Statement

The WAIT statement lets you test one or more bit positions on an input port. You can test the bit position for either a 1 or a 0. The bit positions to be tested are specified by setting 1's in those positions in *n*. If you do not specify *m*, the input port bits are tested for 1's. If you do specify *m*, a 1 in any bit position in *m* (for which there is a 1 bit in *n*) causes WAIT to test for a 0 for that input bit.

When executed, the WAIT statement loops testing those input bits specified by 1's in *n*. If *any one* of those bits is 1 (or 0 if the corresponding bit in *m* is 1), then the program continues with the next statement. Thus WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

> **Note:** It is possible to enter an infinite loop with the WAIT statement. You can do a Ctrl-Break or a System Reset to exit the loop.

Example: To suspend program execution until port 32 receives a 1 bit in the second bit position:

```
100 WAIT 32,2
```

WHILE and WEND Statements

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

Versions:

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

Format: WHILE *expression*

.

.

.

(*loop statements*)

.

.

.

WEND

Remarks: *expression* is any numeric expression.

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

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

WHILE and WEND Statements

Example: This example sorts the elements of the string array A$ into alphabetical order. A$ was defined with J elements.

```
90 'bubble sort array A$
100 FLIPS=1 'force one pass thru loop
110 WHILE FLIPS
115  FLIPS=0
120  FOR I=1 TO J-1
130   IF A$(I)>A$(I+1) THEN
          SWAP A$(I),A$(I+1): FLIPS=1
140  NEXT I
150 WEND
```

WIDTH
Statement

Purpose: Sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.

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

Format: WIDTH *size*

WIDTH *filenum,size*

WIDTH *device,size*

Remarks:

size	is a numeric expression in the range 0 to 255. This is the new width. WIDTH 0 is the same thing as WIDTH 1.
filenum	is a numeric expression in the range 1 to 15. This is the number of a file opened to one of the devices listed below.
device	is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, LPT2:, LPT3:, COM1:, or COM2:.

Depending upon the device specified, the following actions are possible:

WIDTH *size* or WIDTH "SCRN:",*size*
Sets the screen width. Only 40 or 80 column width is allowed.

If the screen is in medium resolution graphics mode (as would occur with a SCREEN 1 statement), WIDTH 80 forces the screen into high resolution (just like a SCREEN 2 statement).

If the screen is in high resolution graphics mode (as would occur with a SCREEN 2 statement), WIDTH 40 forces the screen into medium resolution (like a SCREEN 1 statement).

> **Note:** Changing the screen width causes the screen to be cleared, and sets the border screen color to black.

WIDTH *device,size*
Used as a deferred width assignment for the device. This form of width stores the new width value without actually changing the current width setting. A subsequent OPEN to the device will use this value for width while the file is open. The width does not change immediately if the device is already open.

> **Note:** LPRINT, LLIST, and LIST,"LPTn:" do an implicit OPEN and are therefore affected by this statement.

WIDTH *filenum,size*
The width of the device associated with *filenum* is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1: in Cassette BASIC. Disk and Advanced BASIC also allow LPT2:, LPT3:, COM1: and COM2:.

Any value entered outside of the ranges indicated will result in an "Illegal function call" error. The previous value is retained.

WIDTH has no effect for the keyboard (KYBD:) or cassette (CAS1:).

The width for each printer defaults to 80 when BASIC is started. The maximum width for the IBM 80 CPS Matrix Printer is 132. However, no error is returned for values between 132 and 255.

It is up to you to set the appropriate physical width on your printer. Some printers are set by sending special codes, some have switches. For the IBM 80 CPS Matrix Printer you should use LPRINT CHR$(15); to change to a condensed typestyle when printing at widths greater than 80. Use LPRINT CHR$(18); to return to normal. The IBM 80 CPS Matrix Printer is set up to automatically add a carriage return if you exceed the maximum line length.

Specifying a width of 255 disables line folding. This has the effect of "infinite" width. WIDTH 255 is the default for communications files.

Changing the width for a communications file does not alter either the receive or the transmit buffer; it just causes BASIC to send a carriage return character after every *size* characters.

Changing screen mode affects screen width only when moving between SCREEN 2 and SCREEN 1 or SCREEN 0. See "SCREEN Statement" in this chapter.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
  .
  .
  .
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current printer width to 40 characters per line.

```
SCREEN 1,0   'Set to med-res color graphics
WIDTH 80     'Go to hi-res graphics
WIDTH 40     'Go back to medium res

SCREEN 0,1   'Go to 40x25 text color mode
WIDTH 80     'Go to 80x25 text color mode
```

WRITE
Statement

Purpose: Outputs data on the screen.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WRITE [*list of expressions*]

Remarks: *list of expressions*

is a list of numeric and/or string expressions, separated by commas or semicolons.

If the list of expressions is omitted, a blank line is output. If the list of expressions is included, the values of the expressions are output on the screen.

When the values of the expressions are output, each item is separated from the last by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, BASIC adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

Example: This example shows how WRITE displays numeric and string values.

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

WRITE
Statement

Purpose: Writes data to a sequential file.

Versions:

Cassette	Disk	Advanced	Compiler
***	***	***	***

Format: WRITE *#filenum, list of expressions*

Remarks: *filenum* is the number under which the file was opened for output.

list of expressions
is a list of string and/or numeric expressions, separated by commas or semicolons.

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

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

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

writes the following image to the file.

"CAMERA","93604-1"

A subsequent INPUT # statement, such as:

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

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

APPENDIXES

Contents

APPENDIX A. MESSAGES A-5

APPENDIX B. BASIC DISKETTE INPUT AND OUTPUT B-1

Specifying Filenames B-2

Commands for Program Files B-2
Protected Files B-3

Diskette Data Files - Sequential and Random I/O B-4
Sequential Files B-4
Creating and Accessing a Sequential File B-4
Adding Data to a Sequential File ... B-7
Random Files B-8
Creating a Random File B-9
Accessing a Random File B-10
An Example Program B-12

Performance Hints B-15

APPPENDIX C. MACHINE LANGUAGE SUBROUTINES C-1

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

APPENDIX D. CONVERTING PROGRAMS TO IBM PERSONAL COMPUTER BASIC ... D-1

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

APPENDIX E. MATHEMATICAL FUNCTIONS ... E-1

APPENDIX F. COMMUNICATIONS ... F-1
Opening a Communications File ... F-1
Communication I/O ... F-1
GET and PUT for Communications Files ... F-2
I/O Functions ... F-2
INPUT$ Functions ... F-3
An Example Program ... F-4
Notes on the Program ... F-5

Operation of Control Signals ... F-6
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

APPENDIX G. ASCII CHARACTER CODES G-1

Extended Codes G-6

APPENDIX H. HEXADECIMAL CONVERSION TABLE H-1

APPENDIX I. TECHNICAL INFORMATION AND TIPS I-1
  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

APPENDIX J. GLOSSARY J-1

NOTES

Appendix A. Messages

If BASIC detects an error that causes a program to stop running, an error message is displayed. It is possible to trap and test errors in a BASIC program using the ON ERROR statement and the ERR and ERL variables. (For complete explanations of ON ERROR, ERR and ERL, see "Chapter 4. BASIC Commands, Statements, Functions, and Variables.")

This appendix lists all the BASIC error messages with their associated error numbers.

Number	*Message*
1	**NEXT without FOR** The NEXT statement doesn't have a corresponding FOR statement. It may be that a variable in the NEXT statement does not correspond to any previously executed and unmatched FOR statement variable. Fix the program so the NEXT has a matching FOR.
2	**Syntax error** A line contains an incorrect sequence of characters, such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation. Or, the data in a DATA statement doesn't match the type (numeric or string) of the variable in a READ statement. When this error occurs, the BASIC program editor automatically displays the line in error. Correct the line or the program.

Number *Message*

3 **RETURN without GOSUB**
A RETURN statement needs a previous unmatched GOSUB statement.

Correct the program. You probably need to put a STOP or END statement before the subroutine so the program doesn't "fall" into the subroutine code.

4 **Out of data**
A READ statement is trying to read more data than is in the DATA statements.

Correct the program so that there are enough constants in the DATA statements for all the READ statements in the program.

5 **Illegal function call**
A parameter that is out of range is passed to a system function. The error may also occur as the result of:

- A negative or unreasonably large subscript
- Trying to raise a negative number to a power that is not an integer
- Calling a USR function before defining the starting address with DEF USR
- A negative record number on GET or PUT (file)
- An improper argument to a function or statement
- Trying to list or edit a protected BASIC program
- Trying to delete line numbers which don't exist

Number *Message*

Correct the program. Refer to "Chapter 4. Basic Commands, Statements, Functions, and Variables" for information about the particular statement or function.

6 **Overflow**
The magnitude of a number is too large to be represented in BASIC's number format. Integer overflow will cause execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues.

To correct integer overflow, you need to use smaller numbers, or change to single- or double-precision variables.

> **Note:** If a number is too small to be represented in BASIC's number format, we have an *underflow* condition. If this occurs, the result is zero and execution continues without an error.

7 **Out of memory**
A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex painting.

You may want to use CLEAR at the beginning of your program to set aside more stack space or memory area.

8 **Undefined line number**
A line reference in a statement or command refers to a line which doesn't exist in the program.

Check the line numbers in your program, and use the correct line number.

Number *Message*

9 **Subscript out of range**
You used an array element either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

Check the usage of the array variable. You may have put a subscript on a variable that is not an array, or you may have coded a built-in function incorrectly.

10 **Duplicate Definition**
You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.
- The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default.

Move the OPTION BASE statement to make sure it is executed before you use any arrays; or, fix the program so each array is defined only once.

11 **Division by zero**
In an expression, you tried to divide by zero, or you tried to raise zero to a negative power.

It is not necessary to fix this condition, because the program continues running. Machine infinity with the sign of the

Number	*Message*
	number being divided is the result of the division; or, positive machine infinity is the result of the exponentiation.
12	**Illegal direct** You tried to enter a statement in direct mode which is invalid in direct mode (such as DEF FN). The statement should be entered as part of a program line.
13	**Type mismatch** You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP variables of different types, such as single- and double-precision.
14	**Out of string space** BASIC allocates string space dynamically until it runs out of memory. This message means that string variables caused BASIC to exceed the amount of free memory remaining after housecleaning.
15	**String too long** You tried to create a string more than 255 characters long. Try to break the string into smaller strings.
16	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.

Number *Message*

17 **Can't continue**
You tried to use CONT to continue a program that:

- Halted due to an error,
- Was modified during a break in execution, or
- Does not exist

Make sure the program is loaded, and use RUN to run it.

18 **Undefined user function**
You called a function before defining it with the DEF FN statement.

Make sure the program executes the DEF FN statement before you use the function.

19 **No RESUME**
The program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (The physical end of the program was encountered in the error trapping routine.)

Be sure to include RESUME in your error trapping routine to continue program execution. You may want to add an ON ERROR GOTO 0 statement to your error trapping routine so BASIC displays the message for any untrapped error.

20 **RESUME without error**
The program has encountered a RESUME statement without having trapped an error. The error trapping routine should only be entered when an error occurs or an ERROR statement is executed.

You probably need to include a STOP or END statement before the error trapping routine to prevent the program from "falling into" the error trapping code.

22 **Missing operand**
An expression contains an operator, such as * or OR, with no operand following it.

Make sure you include all the required operands in the expression.

23 **Line buffer overflow**
You tried to enter a line that has too many characters.

Separate multiple statements on the line so they are on more than one line. You might also use string variables instead of constants where possible.

24 **Device Timeout**
BASIC did not receive information from an input/output device within a predetermined amount of time. In Cassette BASIC, this only occurs while the program is trying to read from the cassette or write to the printer.

For communications files, this message indicates that one or more of the signals tested with OPEN "COM... was not found in the specified period of time.

Retry the operation.

25 **Device Fault**
A hardware error indication was returned by an interface adapter.

In Cassette BASIC, this only occurs when a fault status is returned from the printer interface adapter.

25 (cont.) This message may also occur when transmitting data to a communications file. In this case, it indicates that one or more of the signals being tested (specified on the OPEN "COM... statement) was not found in the specified period of time.

26 **FOR without NEXT**
A FOR was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached.

Correct the program so it includes a NEXT statement.

27 **Out of Paper**
The printer is out of paper, or the printer is not switched on.

You should insert paper (if necessary), verify that the printer is properly connected, and make sure that the power is on; then, continue the program.

29 **WHILE without WEND**
A WHILE statement does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

30 **WEND without WHILE**
A WEND is encountered before a matching WHILE was executed.

Correct the program so that there is a WHILE for each WEND.

Number *Message*

50 **FIELD overflow**
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer is encountered while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.

51 **Internal error**
An internal malfunction occurred in BASIC.

Recopy your diskette. Check the hardware and retry the operation. If the error reoccurs, report to your computer dealer the conditions under which the message appeared.

52 **Bad file number**
A statement uses a file number of a file that is not open, or the file number is out of the range of possible file numbers specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename was too long or invalid.

Make sure the file you wanted was opened and that the file number was entered correctly in the statement. Check that you have a valid file specification (refer to "Naming Files" in Chapter 3 for information on file specifications).

Number *Message*

53 **File not found**
A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the diskette in the specified drive.

Verify that the correct diskette is in the drive specified, and that the file specification was entered correctly. Then retry the operation.

54 **Bad file mode**
You tried to use PUT or GET with a sequential file or a closed file; or to execute an OPEN with a file mode other than input, output, append, or random.

Make sure the OPEN statement was entered and executed properly. GET and PUT require a random file.

This error also occurs if you try to merge a file that is not in ASCII format. In this case, make sure you are merging the right file. If necessary, load the program and save it again using the A option.

55 **File already open**
You tried to open a file for sequential output or append, and the file is already opened; or, you tried to use KILL on a file that is open.

Make sure you only execute one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.

57 **Device I/O Error**
An error occurred on a device I/O operation. DOS cannot recover from the error.

When receiving communications data, this error can occur from overrun, framing, break, or parity errors. When you are receiving data with 7 or less data bits, the eighth bit is turned on in the byte in error.

58 **File already exists**
The filename specified in a NAME statement matches a filename already in use on the diskette.

Retry the NAME command using a different name.

61 **Disk full**
All diskette storage space is in use. Files are closed when this error occurs.

If there are any files on the diskette that you no longer need, erase them; or, use a new diskette. Then retry the operation or rerun the program.

62 **Input past end**
This is an end of file error. An input statement is executed for a null (empty) file, or after all the data in a sequential file was already input.

To avoid this error, use the EOF function to detect the end of file.

This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.

Number *Message*

63 **Bad record number**
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

Correct the PUT or GET statement to use a valid record number.

64 **Bad file name**
An invalid form is used for the filename with BLOAD, BSAVE, KILL, NAME, OPEN, or FILES.

Check "Naming Files" in Chapter 3 for information on valid filenames, and correct the filename in error.

66 **Direct statement in file**
A direct statement was encountered while loading or chaining to an ASCII format file. The LOAD or CHAIN is terminated.

The ASCII file should consist only of statements preceded by line numbers. This error may occur because of a line feed character in the input stream. Refer to "Appendix D. Converting Programs to IBM Personal Computer BASIC."

67 **Too many files**
An attempt is made to create a new file (using SAVE or OPEN) when all directory entries on the diskette are full, or when the file specification is invalid.

If the file specification is okay, use a new formatted diskette and retry the operation.

68 **Device Unavailable**
You tried to open a file to a device which doesn't exist. Either you do not have the hardware to support the device (such as

printer adapters for a second or third printer), or you have disabled the device. (For example, you may have used **/C:0** on the BASIC command to start Disk BASIC. That would disable communications devices.)

Make sure the device is installed correctly. If necessary, enter the command:

```
SYSTEM
```

This returns you to DOS where you can re-enter the BASIC command.

69 **Communication buffer overflow**
A communication input statement was executed, but the input buffer was already full.

You should use an ON ERROR statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the **/C:** option when you start BASIC.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. (See the example in "Appendix F. Communications.")
- Use a lower baud rate to transmit and receive.

Number	*Message*
70	**Disk Write Protect** You tried to write to a diskette that is write-protected. Make sure you are using the right diskette. If so, remove the write protection, then retry the operation. This error may also occur because of a hardware failure.
71	**Disk not Ready** The diskette drive door is open or a diskette is not in the drive. Place the correct diskette in the drive and continue the program.
72	**Disk Media Error** The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad. Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.
73	**Advanced Feature** Your program used an Advanced BASIC feature while you were using Disk BASIC. Start Advanced BASIC and rerun your program.
—	**Unprintable error** An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code. Check your program to make sure you handle all error codes which you create.

Appendix B. BASIC Diskette Input and Output

This appendix describes procedures and special considerations for using diskette input and output. It contains lists of the commands and statements that are used with diskette files, and explanations of how to use them. Several sample programs are included to help clarify the use of data files on diskette. If you are new to BASIC or if you're getting diskette-related errors, read through these procedures and program examples to make sure you're using all the diskette statements correctly.

You may also want to refer to the *IBM Personal Computer Disk Operating System* manual for other information on handling diskettes and diskette files.

> **Note:** Most of the information in this appendix about program files and sequential files applies to cassette I/O as well. The cassette cannot be opened in random mode, however.

Specifying Filenames

Filenames for diskette files must conform to DOS naming conventions in order for BASIC to be able to read them. Refer to "Naming Files" in Chapter 3 to be sure you are specifying your diskette files correctly.

Commands for Program Files

The commands which you can use with your BASIC program files are listed below, with a quick description. For more detailed information on any of these commands, refer to "Chapter 4. BASIC Commands, Statements, Functions, and Variables."

SAVE filespec [,A]
Writes to diskette the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD filespec [,R]
Loads the program from diskette into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections, and can access the same data files.

RUN filespec [,R]

RUN *filespec* loads the program from diskette into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the **R** option is included, however, all open data files are kept open.

MERGE filespec

Loads the program from diskette into memory, but does not delete the current contents of memory. The program line numbers on diskette are merged with the line numbers in memory. If two lines have the same number, only the line from the diskette program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.

KILL filespec

Deletes the file from the diskette.

NAME filespec AS filename

Changes the name of a diskette file.

Protected Files

If you wish to save a program in an encoded binary format, use the **P** (protect) option with the SAVE command. For example:

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

A program saved this way cannot be listed, saved, or edited. Since you cannot "unprotect" such a program, you may also want to save an unprotected copy of the program for listing and editing purposes.

Diskette Data Files - Sequential and Random I/O

Two types of diskette data files may be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

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

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

CLOSE
INPUT #
LINE INPUT #
OPEN
PRINT #
PRINT # USING
WRITE #
EOF
INPUT$
LOC
LOF

Creating and Accessing a Sequential File

To create a sequential file and access the data in the file, include the following steps in your program:

1. Open the file for output or append using the OPEN statement.

2. Write data to the file using the PRINT #, WRITE #, or PRINT # USING statements.

3. To access the data in the file, you must close the file (using CLOSE) and reopen it for input (using OPEN).

4. Use the INPUT# or LINE INPUT# statements to read data from the sequential file into the program.

The following are example program lines that demonstrate these steps.

```
100  OPEN "DATA" FOR OUTPUT AS #1 'step 1
200  WRITE #1,A$,B$,C$             'step 2
300  CLOSE #1                     'step 3
400  OPEN "DATA" FOR INPUT AS #1 'also step 3
500  INPUT #1,X$,Y$,Z$             'step 4
```

The above program could also have been written as follows:

```
100  OPEN "O",#1,"DATA"    'step 1
200  WRITE #1,A$,B$,C$     'step 2
300  CLOSE #1              'step 3
400  OPEN "I",#1,"DATA"    'still step 3
500  INPUT #1,X$,Y$,Z$      'step 4
```

Notice that both ways of writing the OPEN statement yield the same results. Look under "OPEN Statement" in Chapter 4 for details of the syntax of each form of OPEN.

The following program, PROGRAM1, is a short program that creates a sequential file, "DATA", from information you enter at the keyboard.

Program 1

```
1 REM PROGRAM1 - create a sequential file
10 OPEN "DATA" FOR OUTPUT AS #1
20 INPUT "NAME";N$
25 IF N$="DONE" THEN CLOSE: END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 WRITE #1,N$,D$,H$
60 PRINT: GOTO 20
RUN
NAME? MICHELANGELO
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

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

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
Ok
```

Now look at PROGRAM2. It accesses the file "DATA" that was created in PROGRAM1 and displays the name of everyone hired in 1978.

Program 2

```
1 REM PROGRAM2 - accessing a sequential file
10 OPEN "DATA" FOR INPUT AS 1
20 INPUT #1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end of file:

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

and change line 40 to GOTO 15. The end of file is indicated by a special character in the file. This character has ASCII code 26 (hex 1A). Therefore, you should not put a CHR$(26) in a sequential file.

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

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

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

The LOC function, when used with a sequential file, returns the number of records that have been written to or read from the file since it was opened. (A record is a 128-byte block of data.) The LOF function returns the number of bytes allocated to the file. This number is always a multiple of 128 (by rounding upward, if necessary).

Adding Data to a Sequential File

If you have a sequential file residing on diskette and later want to add more data to the end of it, you cannot simply open the file for output and start writing data. As soon as you open a sequential file for output, you destroy its current contents. Instead, you should open the file for APPEND. Refer to "OPEN Statement" in Chapter 4 for details.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. For instance, numbers in random files are usually stored on diskette in binary formats, while numbers in sequential files are stored as ASCII characters. Therefore, in many cases random files require less space on diskette than sequential files.

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

Records may be any length up to 32767 bytes. The size of a record is not related to the size of a sector on the diskette (512 bytes). BASIC automatically uses all 512 bytes in a sector for information storage. It does this by both blocking records and spanning sector boundaries (that is, part of a record may be at the end of one sector and the other part at the beginning of the next sector).

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

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

Creating a Random File

The following program steps are required to create a random file.

1. Open the file for random access. The example which follows to illustrate these steps specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

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

4. Write the data from the buffer to the diskette using the PUT statement.

The following lines illustrate these steps:

```
100  OPEN "FILE" AS #1 LEN=32 'step 1
200  FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
                              'step 2
300  LSET N$=X$               'step 3
400  LSET A$=MKS$(AMT)        'still step 3
500  LSET P$=TEL$             'still step 3
600  PUT #1,CODE%             'step 4
```

Note: Do not use a string variable which has been defined in a FIELD statement in an input statement or on the left side of an assignment (LET) statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Look at PROGRAM3. It takes information that is entered at the keyboard and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Program 3

```
1 REM PROGRAM3 - create a random file
10 OPEN "FILE" AS #1 LEN=32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$: PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Accessing a Random File

The following program steps are required to access a random file:

1. Open the file for random access.

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

   Note: In a program that performs both input and output on the same random file, you can usually use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

The following program lines illustrate these steps:

```
100 OPEN "FILE" AS 1 LEN=32    'step 1
200 FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
                               'step 2
300 GET #1,CODE%               'step 3
400 PRINT N$                   'step 4
500 PRINT CVS(A$)              'still step 4
```

PROGRAM4 accesses the random file "FILE" that was created in PROGRAM3. By entering the two-digit code at the keyboard, the information associated with that code is read from the file and displayed.

Program 4

```
1 REM PROGRAM4 - access a random file
10 OPEN "FILE" AS 1 LEN=32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
```

The LOC function, with random files, returns the "current record number." The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

An Example Program

PROGRAM5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 690-750 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 180 and line 320) to determine whether an entry already exists for that part number.

Lines 40-120 display the different inventory functions that the program performs. When you type in the desired function number, line 140 branches to the appropriate subroutine.

Program 5

```
10 REM PROGRAM5 - inventory
20 OPEN "inven.dat" AS #1 LEN=39
30 FIELD #1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
40 PRINT: PRINT"Options:": PRINT
50 PRINT 1,"Initialize File"
60 PRINT 2,"Create a New Entry"
70 PRINT 3,"Display Inventory for One Part"
80 PRINT 4,"Add to Stock"
90 PRINT 5,"Subtract from Stock"
100 PRINT 6,"List Items Below Reorder Level"
110 PRINT 7,"End Application"
120 PRINT: PRINT: INPUT "Choice";CHOICE
130 IF (CHOICE<1) OR (CHOICE>7) THEN PRINT "Bad Choice Number"
    : GOTO 40
140 ON CHOICE GOSUB 690, 160, 300, 390, 470, 590, 760
150 GOTO 120
160 REM      build new entry
170 GOSUB 670
180 IF ASC(F$)<>255 THEN INPUT "Overwrite";A$:
    IF A$<>"y" AND A$<>"Y" THEN RETURN
190 LSET F$=CHR$(0)
200 INPUT "Description";DESC$
210 LSET D$=DESC$
```

```
220 INPUT "Quantity in stock";Q%
230 LSET Q$=MKI$(Q%)
240 INPUT "Reorder level";R%
250 LSET R$=MKI$(R%)
260 INPUT "Unit price";P
270 LSET P$=MKS$(P)
280 PUT #1,PART%
290 RETURN
300 REM       display entry
310 GOSUB 670
320 IF ASC(F$)=255 THEN PRINT "Null entry": RETURN
330 PRINT USING "Part number ###";PART%
340 PRINT D$
350 PRINT USING "Quantity on hand #####";CVI(Q$)
360 PRINT USING "Reorder level #####";CVI(R$)
370 PRINT USING "Unit price $$##.##";CVS(P$)
380 RETURN
390 REM       add to stock
400 GOSUB 670
410 IF ASC(F$)=255 THEN PRINT"Null entry":RETURN
420 PRINT D$:INPUT "Quantity to add";A%
430 Q%=CVI(Q$)+A%
440 LSET Q$=MKI$(Q%)
450 PUT #1,PART%
460 RETURN
470 REM       remove from stock
480 GOSUB 670
490 IF ASC(F$)=255 THEN PRINT "Null entry": RETURN
500 PRINT D$
510 INPUT "Quantity to subtract";S%
520 Q%=CVI(Q$)
530 IF (Q%-S%)<0 THEN PRINT"Only";Q%;"in stock": GOTO 510
540 Q%=Q%-S%
550 IF Q%=<CVI(R$) THEN PRINT "Quantity now";Q%;
      ", Reorder level";CVI(R$)
560 LSET Q$=MKI$(Q%)
570 PUT #1,PART%
580 RETURN
590 REM       list items below reorder level
600 FOR I=1 TO 100
610 GET #1,I
620 IF ASC(F$)=255 THEN 640
630 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" Quantity";CVI(Q$)
       TAB(50) "Reorder level";CVI(R$)
640 NEXT I
650 RETURN
```

```
660 REM   get part record
670 INPUT "Part number";PART%
680 IF PART%<1 OR PART%>100
      THEN PRINT "Bad part number": GOTO 670
      ELSE GET #1,PART%: RETURN
690 REM initialize file
700 INPUT "Are you sure";B$: IF B$<>"Y" AND B$<>"y"
     THEN RETURN
710 LSET F$=CHR$(255)
720 FOR I=1 TO 100
730 PUT #1,I
740 NEXT I
750 RETURN
760 REM       end application
770 CLOSE: END
```

Performance Hints

- If you do not use random files, specify **/S:0** on the BASIC command when you start BASIC. This will save 128 bytes times the number of files specified in the /F: option.

- BASIC sets up three files by default. If you use less than three, set /F:***files*** when you start BASIC with the BASIC command. Note that the screen, keyboard, and printer do not count as files unless you explicitly OPEN them.

- Sequential files use a buffer of 128 bytes. Random files also default to a buffer of 128 bytes, but this can be overridden with the **/S:** option on the BASIC command. There is no advantage to setting **/S:** to a number greater than the largest record length on any of your random files. However, the combination of a record length of 512 and **/S:512** gives improved performance since the diskette sector size is 512 bytes.

  If you want to do sequential I/O, but still want improved performance, you can use random files to do "pseudo-sequential" I/O. For example:

  ```
  1 ' example 1A
  10 OPEN "ABC" FOR OUTPUT AS #1
  20 FOR I=1 TO 3000
  30 PRINT #1,"MELH"
  40 NEXT
  50 CLOSE #1: END
  ```

  This example (1A) uses regular sequential I/O to create a file with 3000 items in it.

```
1 ' example 1B
10 OPEN "ABC" FOR INPUT AS #1
20 OPEN "DEF" FOR OUTPUT AS #2
30 IF EOF(1) THEN CLOSE: END
40 INPUT #1,A$
50 PRINT #2,A$
60 GOTO 30
70 END
```

This example (1B) copies the sequential file "ABC", which we just created, to a file named "DEF".

For the next examples we will perform the same task using "pseudo-sequential" I/O.

```
1 ' example 2A
10 OPEN "PQR" AS #1 LEN=512
15 ON ERROR GOTO 90
20 FOR I=1 TO 3000
30 PRINT #1,"MELH"
40 NEXT
45 PRINT #1,"/eof"
50 ON ERROR GOTO 0: PUT #1: CLOSE
60 END
90 PUT #1: RESUME
```

This example (2A) creates a file with 3000 items using random I/O. This is a "pseudo-sequential" file.

```
1 ' example 2B
10 OPEN "PQR" AS #1 LEN=512
20 OPEN "XYZ" AS #2 LEN=512
30 ON ERROR GOTO 80
40 GET #1
50 INPUT #1,A$
60 PRINT #2,A$
70 IF A$<>"/eof" THEN 50 ELSE
       ON ERROR GOTO 0: PUT #2: CLOSE: END
80 IF ERL=50 THEN GET #1: RESUME
       ELSE PUT #2: RESUME
```

This final example copies the "pseudo-sequential" file created in the previous example to a new "pseudo-sequential" file named "XYZ". It takes about half as long to run as the example using sequential I/O.

The technique used in these examples involves detection of the "FIELD overflow" error (error 50) and doing the appropriate GET or PUT to purge the buffer (line 90 in example 2A and line 80 in example 2B). Note also that a dummy end of file must be written ("/eof" in the example) and checked for during input. Also, the INPUT and PRINT statements use only single variables, rather than a list of variables.

This technique is useful only when you have more than one file open at a time.

NOTES

Appendix C. Machine Language Subroutines

This appendix describes how BASIC interfaces with machine language subroutines. In particular, it describes:

- How to allocate memory for the subroutines
- How to get the machine language subroutine into memory
- How to call the subroutine from BASIC and pass parameters to it

This appendix is intended to be used by an experienced machine language programmer.

Reference Material

Rector, Russell and Alexy, George. *The 8086 Book*. Osborne/McGraw-Hill, Berkeley, California, 1980. (includes the 8088)

Intel Corporation Literature Department. *The 8086 Family User's Manual*, 9800722.
3065 Bowers Avenue, Santa Clara, CA 95051.

IBM Corporation Personal Computer library. *Macro-Assembler*. Boca Raton, FL 33432.

IBM Corporation Personal Computer library. *Technical Reference*. Boca Raton, FL 33432.

Setting Memory Aside for Your Subroutines

BASIC normally uses all memory available from its starting location up to a maximum of 64K-bytes. This BASIC workarea contains your BASIC program and data, along with the interpreter workarea and BASIC's stack. You may allocate memory space for machine language subroutines either inside or outside of this BASIC 64K workarea. Where you decide to put the routines depends on the total amount of available memory and the size of the applications to be loaded.

Your system needs more than 64K-bytes of memory if you want to put your machine language subroutines outside BASIC's 64K workarea. If you are using Disk or Advanced BASIC, DOS takes up approximately 12K-bytes, and BASIC takes up another 10K-bytes, so you need at least a 96K-byte system in order for there to be room outside the BASIC workarea for the machine language subroutines.

Outside the BASIC Workarea: If your system has enough memory that you can put your subroutines outside the BASIC 64K-byte workarea, you don't have to do anything to reserve that area. You use the DEF SEG statement to address the external subroutine area outside the BASIC workarea.

For example, in a 96K-byte system, to specify an address in the upper 4K-bytes of memory, you could use:

```
110 DEF SEG=&H1700
```

This statement specifies a segment starting at hexadecimal location 17000 (92K).

Inside the BASIC Workarea: In order to keep BASIC from writing over your subroutines in memory, use either:

- The CLEAR statement, which is available in all versions of BASIC
- The /M: option on the BASIC command to start Disk and Advanced BASIC from DOS

Only the highest memory locations can be set aside for subroutines. For example, to reserve the highest 4K-byte area of BASIC's 64K-byte workarea for your machine language subroutines, you could use:

```
10 CLEAR ,&HF000
```

or start BASIC with the DOS command:

```
BASIC /M:&HF000
```

Either of these statements restricts the size of the BASIC workarea to hex F000 (60K) bytes, so you can use the uppermost 4K-bytes for machine language subroutines.

Getting the Subroutine Code into Memory

The following are offered as suggestions as to how machine language subroutines can be loaded. We don't describe all possible situations.

Two common ways to get a machine language program into memory are:

- Poking it into memory from your BASIC program
- Loading it from a file on diskette or cassette

Poking a Subroutine into Memory

You can code relatively short subroutines in machine language and use the POKE statement to put the code into memory. In this way, the subroutine actually becomes a part of your BASIC program. One way to do this is:

1. Determine the machine code for your subroutine.
2. Put the hex value (&Hxx format) of each byte of the code into DATA statements.
3. Execute a loop which reads each data byte, and then pokes it into the area you've selected for the subroutine (see the preceding discussion).
4. After the loop is complete, the subroutine is loaded. If you are going to call the subroutine using the USR function, then you must execute a DEF USR statement to define the entry address of the subroutine; if you are going to call the subroutine using the CALL statement, you must set the value of the subroutine variable to the subroutine's entry address.

For example:

```
Ok
10 DEFINT A-Z
20 DEF SEG=&H1700
30 FOR I=0 TO 21
40 READ J
50 POKE I,J
60 NEXT
70 SUBRT=0
80 A=2:B=3:C=0
90 CALL SUBRT(A,B,C)
100 PRINT C
110 END
120 DATA &H55,&H8B,&HEC,&H8B,&H76,&H0A
130 DATA &H8B,&H04,&H8B,&H76,&H08
140 DATA &H03,&H04,&H8B,&H7E,&H06
150 DATA &H89,&H05,&H5D,&HCA,&H06,&H00
RUN
 5
Ok
```

Loading the Subroutine from a File

You use the BASIC BLOAD command to load a memory image file directly into memory. The memory image can be a machine language subroutine which was saved using the BSAVE command. Of course, that leads to the question of how the subroutine got there in the first place. The machine language subroutine may be an executable file which was created by the linker from DOS, and which was placed into memory using DEBUG. DEBUG and the linker are explained in the *IBM Personal Computer Disk Operating System* manual.

The following is a suggested way to use BLOAD to get such a machine language subroutine into memory:

1. Use the linker to produce an .EXE file of your routine (let's call it ASMROUT.EXE) so it will load at the HIGH end of memory.

2. Load BASIC under DEBUG by entering:

```
DEBUG BASIC.COM
```

3. Display the registers (use the R command) to find out where BASIC was put in memory. Record the values contained in the registers (CS, IP, SS, SP, DS, ES) for later reference.

4. Use DEBUG to load the .EXE file (your subroutine) into HIGH memory, where it will overlay the transient portion of COMMAND.COM.

```
N ASMROUT.EXE
L
```

5. Display the registers (use the R command) to find out where the subroutine was placed in memory. Record the values contained in the CS and IP registers for later use.

6. Reset the registers (use the R command) back to the values they contained when BASIC.COM was originally loaded, using the values noted in step 3.

7. Use the G command to branch to the BASIC entry point and to set breakpoints (if desired) in the machine language subroutine.

8. When BASIC prompts, load your BASIC application program and edit the DEF SEG and either the DEF USR statement or the value of the CALL variable to correspond with the location of the subroutine as determined when you loaded the subroutine in step 5.

   - Use the previously recorded value in the CS register for DEF SEG
   - Use the previously recorded value in the IP register for the DEF USR or the variable value of the CALL

9. In direct mode in BASIC, enter a BSAVE command to save the subroutine area. Use the starting location defined by the CS and IP registers when the subroutine was loaded in step 5, and the code length printed on the assembler listing or LINK map. (Refer to "BSAVE Command" in Chapter 4.)

10. Edit your BASIC application program so it contains a BLOAD statement after the DEF SEG that sets the proper value of CS for the subroutine.

Note: If the machine language routine is self-relocatable, BLOAD can be used to put the subroutine some place other than where the linker originally placed it. If you make such a change, be sure to make a corresponding change to the DEF SEG statement associated with the call so that BASIC can find the subroutine at execution time.

Some suggestions for alternate locations for the subroutine are:

- An unused screen buffer
- An unused file buffer (located with VARPTR(#*f*))
- A string variable area located with VARPTR(*stringvar*)

(See 'BLOAD Command" and "VARPTR Function" in Chapter 4.)

11. Save the resulting modified BASIC application.

Some Notes on Using DEBUG with BASIC: When you run BASIC under DEBUG, BASIC is loaded after DEBUG in memory, so DEBUG is not written over if you load a large BASIC program. If you set breakpoints in your machine language subroutine, they return you to DEBUG. The SYSTEM command also returns you from BASIC to DEBUG.

Calling the Subroutine from Your BASIC Program

All versions of BASIC have two ways to call machine language subroutines: the USR function, and the CALL statement. This section describes the use of both USR and CALL.

Common Features of CALL and USR

Whether you call your machine language subroutines with CALL or with the USR function, you must keep the following things in mind:

Entering the Subroutine

- At entry, the segment registers DS, ES, and SS are all set to the same value, the address of BASIC's data space (the default for DEF SEG).
- At entry, the code segment register, CS, contains the current value specified in the latest DEF SEG. If DEF SEG has not been specified, or if the latest DEF SEG did not specify an override value, the value in CS is the same as in the other three segment registers.

String Arguments

- If an input argument is a string, the value received in the argument is the address of a three-byte area called the *string descriptor*:

  1. Byte 0 of the string descriptor contains the length of the string (0 to 255).
  2. Byte 1 of the string descriptor contains the lower 8 bits of the offset of the string in BASIC's data space.

3. Byte 2 of the string descriptor contains the higher 8 bits of the offset of the string in BASIC's data space.

The string itself is pointed to by the last two bytes of the string descriptor.

Warning:
The subroutine must not change the contents of any of the three bytes of the *string descriptor.*

The subroutine may change the *content* of the string itself, but not its *length*.

If the subroutine changes a string, be aware that this may *modify your program.* The following example may change the string "TEXT"in the BASIC program.

```
A$ = "TEXT"
CALL SUBRT(A$)
```

However, the next example does not modify the program, because the string concatenation causes BASIC to copy the string into the string space where it may be safely changed without affecting the original text.

```
A$ = "BASIC" +""
CALL SUBRT(A$)
```

Returning from the Subroutine

- The return to BASIC must be by an inter-segment RET instruction. (The subroutine is a FAR procedure.)

- At exit, all segment registers and the stack pointer, SP, must be restored. All other registers (and flags) may be altered.

- The stack pointer, at entry, indicates a stack that has only 16 bytes (eight words) available for use by the subroutine. If more stack space is needed, the subroutine must set up its own stack segment and stack pointer. You should make sure that the location of the current stack is recorded so its pointer can be restored just prior to return.
- If interrupts were disabled by the subroutine, they should be enabled prior to return.

CALL Statement

Machine language subroutines may be called using the BASIC CALL statement. The format of the CALL statement is:

CALL *numvar* [(*variable list*)]

numvar	is the name of a numeric variable. Its value is the offset, from the segment set by DEF SEG, that is the starting point in memory of the subroutine being called.
variable list	contains the variables, separated by commas, that are to be passed as arguments to the routine. (The arguments cannot be constants.)

Execution of a CALL statement causes the following:

1. For each variable in the variable list, the variable's location is pushed onto the stack. The location is specified as a two-byte offset into BASIC's data segment (the default DEF SEG).

2. The return address specified in the CS register and the offset are pushed onto the stack.

3. Control is transferred to the machine language routine using the segment address specified in the last DEF SEG statement and the offset specified by the value of *numvar*.

Notes for the CALL Statement

- You can return values to BASIC through the arguments by changing the values of the variables in the argument list.

- If the argument is a string, the offset for the argument points to the three-byte *string descriptor* as explained previously.

- The called routine must know how many arguments were passed. Parameters are referenced by adding a positive offset to BP after the called routine moves the current stack pointer into BP. The first instructions in the subroutine should be:

```
PUSH BP     ;SAVE BP
MOV BP,SP   ;MOVE SP TO BP
```

The offset into the stack of any one particular argument is calculated as follows:

offset from BP = $2*(n-m)+6$

where:

n is the total number of arguments passed.

m is the position of the specific argument in the argument list of the BASIC CALL statement (*m* may range from 1 to *n*).

Example: The following example adds the values in A% and B% and stores the result in C%:

The following statements are in BASIC:

```
100 A%=2: B%=3
200  DEF SEG=&H27E0
250  BLOAD "SUBRT.EXE",0
300  SUBRT=0
400  CALL SUBRT (A%,B%,C%)
500 PRINT C%
```

> **Note:** Line 200 sets the segment to location hex 27E00. SUBRT is set to 0 so that the call to SUBRT executes the subroutine at location &H27E00.

The following statements are in IBM Personal Computer Macro-Assembler source code:

```
CSEG    SEGMENT
        ASSUME   CS:CSEG
SUBRT   PROC     FAR
        PUSH BP          ;SAVE BP
        MOV BP,SP        ;SET BASE PARM LIST
        MOV SI,[BP]+10   ;GET ADDR PARM A
        MOV AX,[SI]      ;GET VALUE OF A
        MOV SI,[BP]+8    ;GET ADDR PARM B
        ADD AX,[SI]      ;ADD VALUE B TO REG
        MOV DI,[BP]+6    ;GET ADDR PARM C
        MOV [DI],AX      ;PASS BACK SUM
        POP BP           ;RESTORE BP
        RET 6            ;FAR RETURN TO BASIC
SUBRT   ENDP
CSEG    ENDS
        END
```

> **Note:** When you call a routine using the CALL statement, the routine must return with a RET *n*, where *n* is 2 times the number of arguments in the variable list. This is necessary to adjust the stack to the point at the start of the calling sequence.

As another example:

```
10 DEFINT A-Z
100 DEF SEG=&H1800
110 BLOAD "SUBRT.EXE",0
120 SUBRT=0
130 CALL SUBRT (A,B$,C)
```

The following sequence of Macro-Assembler code shows how the arguments (including the address of a string descriptor) are passed and accessed, and how the result is stored in variable C:

```
PUSH   BP          ;SAVE BP
MOV    BP,SP       ;GET CURRENT STK POSITION INTO BP
MOV    BX,[BP]+8   ;GET ADDR OF B$ STRING DESCRIPTOR
MOV    CL,[BX]     ;GET LENGTH OF B$ INTO CL
MOV    DX,1[BX]    ;GET ADDR OF B$ TEXT INTO DX
  .
  .
  .
MOV    SI,[BP]+10  ;GET ADDR OF A INTO SI
MOV    DI,[BP]+6   ;GET ADDR OF C INTO DI
MOVS   WORD        ;STORE VARIABLE A INTO C
POP    BP          ;RESTORE BP
RET    6           ;RESTORE STACK, RETURN
END
```

Warning:
It is entirely up to you to make sure that the arguments in the CALL statement match in number, type, and length with the arguments expected by the subroutine.

In the preceding example, the instruction **MOVS WORD** copies only two bytes because variables A and C are integers. However, if A and C are single-precision, four bytes must be copied; if A and C are double-precision, eight bytes must be copied.

USR Function Calls

The other way to call machine language subroutines from BASIC is with the USR function. The format of the USR function is:

USR[*n*](*arg*)

n must be a single digit in the range 0 through 9.

arg is any numeric expression or a string variable name.

n specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If *n* is omitted, USR0 is assumed. The address specified in the DEF USR statement determines the starting address of the subroutine. Even if the subroutine does not require an argument, a dummy argument must still be supplied.

When the USR function is called, register AL contains a value that specifies the type of argument that was supplied. The value in AL will be one of the following:

Value in AL	**Type of Argument**
2	Two-byte integer (two's complement)
3	String
4	Single-precision number
8	Double-precision number

If the argument is a string, the DX register points to the three-byte string descriptor. (See "Common Features of CALL and USR," described previously.)

If the argument is a number and not a string, the value of the argument is placed in the Floating Point Accumulator (FAC), which is an eight-byte area in BASIC's data space. In this case, the BX register contains the offset within the BASIC data space to the fifth byte of the eight-byte FAC. For the following examples, assume that the FAC is in bytes hex 49F through hex 4A6; that is, BX contains hex 4A3:

If the argument is an integer:

- Hex 4A4 contains the upper 8 bits of the argument.
- Hex 4A3 contains the lower 8 bits of the argument.

If the argument is a single-precision number:

- Hex 4A6 contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa. Hex 4A5 contains the highest 7 bits of the mantissa with the leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive; 1 = negative).
- Hex 4A4 contains the middle 8 bits of the mantissa.
- Hex 4A3 contains the lowest 8 bits of the mantissa.

If the argument is a double-precision number:

- Hex 4A3 through hex 4A6 are the same as described under single-precision floating-point number in the preceding paragraph.
- Hex 49F through Hex 4A2 contain four more bytes of the mantissa (hex 49F contains the lowest 8 bits).

Usually, the value returned by a USR function is the same type (integer, string, single-precision, or double-precision) as the argument that was passed to it. However, a numerical argument of the function, regardless of its type, may be forced to an integer value by calling the FRCINT routine to get the integer equivalent of the argument placed into register BX.

If the value being returned by the function is to be an integer, place the resulting value into the BX register. Then make a call to MAKINT just prior to the inter-segment return. This passes the value back to BASIC by placing it into the FAC.

The methods for accessing FRCINT and MAKINT are shown in the following example:

```
100 DEF SEG=&H1800
120 BLOAD "SUBRT.EXE",0
130 DEF USR0=0
140 X = 5 'Note that X is single-precision
150 Y = USR0(X)
160 PRINT Y
```

At location 1800:0 (segment:offset), the following Macro-Assembler language routine has been loaded. The routine doubles the argument passed and returns an integer result:

```
RSEG       SEGMENT AT 0F600H ;BASE OF BASIC ROM
           ORG   3       ;OFFSET TO FORCE INTEGER
  FRCINT   LABEL FAR
           ORG   7       ;OFFSET TO MAKE INTEGER
  MAKINT   LABEL FAR
RSEG      ENDS
   .
   .
   .
CSEG       SEGMENT
USRPRG     PROC  FAR     ;ENTRY POINT
           CALL  FRCINT ;FORCE ARG IN FAC INTO [BX]
           ADD   BX,BX   ;[BX] = [BX] * 2
           CALL  MAKINT ;PUT INT RSLT IN BX INTO FAC
           RET           ;INTER-SEGMENT RETURN TO BASIC
USRPRG     ENDP
CSEG       ENDS
```

Note: FRCINT and MAKINT perform inter-segment returns. You should make sure that the calls to FRCINT and MAKINT are defined by a FAR procedure.

NOTES

Appendix D. Converting Programs to IBM Personal Computer BASIC

Since IBM Personal Computer BASIC is very similar to many microcomputer BASICs, the IBM Personal Computer will support programs written for a wide variety of microcomputers. If you have programs written in a BASIC other than IBM Personal Computer BASIC, some minor adjustments may be necessary before running them with IBM Personal Computer BASIC. Here are some specific things to look for when converting BASIC programs.

File I/O

In IBM Personal Computer BASIC, you read and write information to a file on diskette or cassette by opening the file to associate it with a particular file number; then using particular I/O statements which specify that file number. I/O to diskette and cassette files is implemented differently in some other BASICs. Refer to the section in Chapter 3 called "Files," and to "OPEN Statement" in Chapter 4 for more specific information.

Also, in IBM Personal Computer BASIC, random file records are automatically blocked as appropriate to fit as many records as possible in each sector.

Graphics

How you draw on the screen varies greatly between different BASICs. Refer to the discussion of graphics in Chapter 3 for specific information about IBM Personal Computer graphics.

IF...THEN

The IF statement in IBM Personal Computer BASIC contains an optional ELSE clause, which is performed when the expression being tested is false. Some other BASICs do not have this capability. For example, in another BASIC you may have:

```
10 IF A=B THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

This sequence of code will still function correctly in IBM Personal Computer BASIC, but it may also be conveniently recoded as:

```
10 IF A=B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

IBM Personal Computer BASIC also allows multiple statements in both the THEN and ELSE clauses. This may cause a program written in another BASIC to perform differently. For example:

```
10 IF A=B THEN GOTO 100 : PRINT "NOT EQUAL"
20 REM CONTINUE
```

In some other BASICs, if the test A=B is false, control branches to the next *statement*; that is, if A is not equal to B, "NOT EQUAL" is printed. In IBM Personal Computer BASIC, both GOTO 100 and PRINT "NOT EQUAL" are considered to be part of the THEN clause of the IF statement. If the test is false, control continues with the next program *line*; that is, to line 20 in this example. PRINT "NOT EQUAL" can never be executed.

This example can be recoded in IBM Personal Computer BASIC as:

```
10 IF A=B THEN 100 ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

Line Feeds

In other BASICs, when you enter a line feed, a line feed character is actually inserted into the text. On the IBM Personal Computer, entering a line feed will pad the rest of the display line with spaces — it does not insert the line feed character. If you try to load a program with line feed characters in it, you will get a "Direct statement in file" error.

Logical Operations

In IBM Personal Computer BASIC, logical operations (NOT, AND, OR, XOR, IMP, and EQV) are performed bit-by-bit on integer operands to produce an integer result. In some other BASICs, the operands are considered to be simple "true" (non-zero) or "false" (zero) values, and the result of the operation is either true or false. As an example of this difference, consider this small program:

```
10 A=9: B=2
20 IF A AND B THEN PRINT "BOTH A AND B ARE TRUE"
```

This example in another BASIC will perform as follows: A is non-zero, so it is true; B is also non-zero, so it is also true; because both A and B are true, A AND B is true, so the program prints **BOTH A AND B ARE TRUE.**

However, IBM Personal Computer BASIC calculates it differently: A is 1001 in binary form, and B is 0010 in binary form, so A AND B (calculated bit-by-bit) is 0000, or zero; zero indicates false, so the message is *not* printed, and the program continues with the next line.

This can affect not only tests made in IF statements, but calculations as well. To get similar results, recode logical expressions like the following:

```
10 A=9: B=2
20 IF (A<>0) AND (B<>0)
      THEN PRINT "BOTH A AND B ARE TRUE"
```

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. IBM Personal Computer BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With IBM Personal Computer BASIC, be sure all statements on a line are separated by a colon (:).

PEEKs and POKEs

Many PEEKs and POKEs are dependent on the particular computer you are using. You should examine the *purpose* of the PEEKs and POKEs in a program in another BASIC, and translate the statement so it performs the same function on the IBM Personal Computer.

Relational Expressions

In IBM Personal Computer BASIC, the value returned by a relational expression, such as A>B, is either -1, indicating the relation is true, of 0, indicating the relation is false. Some other BASICs return a positive 1 to indicate true. If you use the value of a relational expression in an arithmetic calculation, the results are likely to be different from what you want.

Remarks

Some BASICs allow you to add remarks to the end of a line using the exclamation point (!). Be sure to change this to a single quote (') when converting to IBM Personal Computer BASIC.

Rounding of Numbers

IBM Personal Computer BASIC rounds single- or double-precision numbers when it requires an integer value. Many other BASICs truncate instead. This can change the way your program runs, because it affects not only assignment statements (for example, I%=2.5 results in I% equal to 3), but also affects function and statement evaluations (for example, 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). Note in particular that rounding may cause IBM Personal Computer BASIC to select a different element from an array than another BASIC — possibly one that is out of range!

Sounding the Bell

Some BASICs require PRINT CHR$(7) to send an ASCII bell character. In IBM Personal Computer BASIC, you may replace this statement with BEEP, although it is not required.

String Handling

String Length: Since strings in IBM Personal Computer BASIC are all variable length, you should delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the IBM Personal Computer BASIC statement DIM A$(J).

Concatenation: Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for IBM Personal Computer BASIC string concatenation.

Substrings: In IBM Personal Computer BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from positin I to position J, must be changed as follows:

Other BASIC	IBM Personal Computer BASIC
`X$=A$(I)`	`X$=MID$(A$,I,1)`
`X$=A$(I,J)`	`X$=MID$(A$,I,J-I+1)`

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	IBM Personal Computer BASIC
`A$(I)=X$`	`MID$(A$,I,1)=X$`
`A$(I,J)=X$`	`MID$(A$,I,J-I+1)=X$`

Use of Blanks

Some BASICs allow statements with no separation of keywords:

```
20FORI=1TOX
```

With IBM Personal Computer BASIC be sure all keywords are separated by a space:

```
20 FOR I=1 TO X
```

Other

The BASIC language on another computer may be different from the IBM Personal Computer BASIC in other ways than those listed here. You should become familiar with IBM Personal Computer BASIC as much as possible in order to be able to appropriately convert any function you may require.

NOTES

Appendix E. Mathematical Functions

Functions that are not intrinsic to IBM Personal Computer BASIC may be calculated as follows.

Function	Equivalent
Logarithm to base B	LOGB(X) = LOG(X)/LOG(B)
Secant	SEC(X) = 1/COS(X)
Cosecant	CSC(X) = 1/SIN(X)
Cotangent	COT(X) = 1/TAN(X)
Inverse sine	ARCSIN(X) = ATN(X/SQR(1-X*X))
Inverse cosine	ARCCOS(X) = 1.570796 -ATN(X/SQR(1-X*X))
Inverse secant	ARCSEC(X) = ATN(SQR(X*X-1)) +(X<0)*3.141593
Inverse cosecant	ARCCSC(X) = ATN(1/SQR(X*X-1)) +(X<0)*3.141593
Inverse cotangent	ARCCOT(X) = 1.57096-ATN(X)
Hyperbolic sine	SINH(X) = (EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X) = (EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X) = (EXP(X)-EXP(-X)) /(EXP(X)+EXP(-X))
Hyperbolic secant	SECH(X) = 2/(EXP(X)+EXP(-X))
Hyperbolic cosecant	CSCH(X) = 2/(EXP(X)-EXP(-X))
Hyperbolic cotangent	COTH(X) = (EXP(X)+EXP(-X)) /(EXP(X)-EXP(-X))
Inverse hyperbolic sine	ARCSINH(X) = LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X) = LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent	ARCTANH(X) = LOG((1+X)/(1-X))/2
Inverse hyperbolic secant	ARCSECH(X) = LOG((1+SQR(1-X*X))/X)
Inverse hyperbolic cosecant	ARCCSCH(X) = LOG((1+SGN(X) *SQR(1+X*X))/X)
Inverse hyperbolic cotangent	ARCCOTH(X) = LOG((X+1)/(X-1))/2

If you use these functions, a good way to code them would be with the DEF FN statement. For example, instead of coding the formula for inverse hyperbolic sine each time you need it, you could code:

```
DEF FNARCSINH(X) = LOG(X+SQR(X*X+1))
```

in one place, then refer to it as

```
FNARCSINH(Y)
```

each time you need it.

Appendix F. Communications

This appendix describes the BASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

Opening a Communications File

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. Refer to "OPEN "COM... Statement" in Chapter 4.

Communication I/O

Since each communications adapter is opened as a file, all input/output statements that are valid for diskette files are valid for communications.

Communications sequential input statements are the same as those for diskette files. They are:

INPUT #
LINE INPUT #
INPUT$

Communications sequential output statements are the same as those for diskette files, and are:

PRINT #
PRINT # USING
WRITE #

Refer to the INPUT and PRINT sections for details of coding syntax and usage.

GET and PUT for Communications Files

GET and PUT are only slightly different for communications files than for diskette files. They are used for fixed length I/O from or to the communications file. In place of specifying the record number to be read or written, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement. Refer to the GET and PUT sections in Chapter 4.

I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates of 1200 bps or higher, it may be necessary to suspend character transmission from the other computer long enough to "catch up." This can be done by sending XOFF (CHR$(19)) to the other computer and XON (CHR$(17)) when ready to resume. XOFF tells the other computer to stop sending, and XON tells it it can start sending again.

> **Note:** This is a commonly used convention, but it is not universal. It depends on the protocol implemented between you and the other computer or peripheral.

Disk and Advanced BASIC provide three functions which help in determining when an "overrun" condition may occur. These are:

LOC(f)	Returns the number of characters in the input buffer waiting to be read. If the number is greater than 255, LOC returns 255.
LOF(f)	Returns the amount of free space in the input buffer. This is the same as *n*-LOC(f), where *n* is the size of the communictions buffer as set by the /C: option on the BASIC command. The default for *n* is 256.

EOF(f) Returns true (-1) if the input buffer is empty; false (0) if there are any characters waiting to be read.

Note: A "Communication buffer overflow" can occur if a read is attempted after the input buffer is full (that is, when LOF(f) returns 0).

INPUT$ Function

The INPUT$ function is preferred over the INPUT # and LINE INPUT # statements when reading communications files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma (,) or carriage return is seen. LINE INPUT # stops when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$(*n,f*) will return *n* characters from the #*f* file. The following statements are efficient for reading a communications file:

```
110 WHILE NOT EOF(1)
120 A$=INPUT$(LOC(1),#1)
 .
 .
 .
(process data returned in A$)
 .
 .
 .
190 WEND
```

These statements return the characters in the buffer into A$ and process them, as long as there are characters in the input buffer. If there are more than 255 characters in the buffer, only 255 will be returned at a time to prevent "String overflow." Further, if this is the case, EOF(1) is false and input continues until the input buffer is empty. Simple, concise, and fast.

In order to process characters quickly, you should avoid, if possible, examining every character as you receive it. If you are looking for special characters (such as control characters), you can use the INSTR function to find them in the input string.

An Example Program

The following program enables the IBM Personal Computer to be used as a conventional "dumb" terminal in a full duplex mode. This program assumes a 300 bps line and an input buffer of 256 bytes (the /C: option was not used on the BASIC command).

```
10 REM   dumb terminal example
20 'set screen to black and white text mode
30 '     and set width to 40
40 SCREEN 0,0: WIDTH 40
50 'turn off soft key display; clear screen;
60 '    make sure all files are closed
70 KEY OFF: CLS: CLOSE
80 'define all numeric variables as integer
90 DEFINT A-Z
100 'define true and false
110 FALSE=0: TRUE= NOT FALSE
120 'define the XON, XOFF characters
130 XOFF$=CHR$(19): XON$=CHR$(17)
140 'open communications to file number 1,
150 ' 300 bps, EVEN parity, 7 data bits
160 OPEN "COM1:300,E,7" AS #1
170 'use screen as a file, just for fun
180 OPEN "SCRN:" FOR OUTPUT AS 2
190 'turn cursor on
200 LOCATE ,,1
400 PAUSE=FALSE: ON ERROR GOTO 9000
490 '
```

```
500 'send keyboard input to com line
510 B$=INKEY$: IF B$<>"" THEN PRINT #1,B$;
520 'if no chars in com buffer, check key in
530 IF EOF(1) THEN 510
540 'if buffer more than 1/2 full, then
550 '   set PAUSE flag to say input suspended,
560 '   send XOFF to host to stop transmission
570 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
580 'read contents of com buffer
590 A$=INPUT$(LOC(1),#1)
600 'get rid of linefeeds to avoid double spaces
610 '   when input displayed on screen
620 LFP=0
630 LFP=INSTR(LFP+1,A$,CHR$(10)) 'look for LF
640 IF LFP>0 THEN MID$(A$,LFP,1)=" ": GOTO 630
650 'display com input, and check for more
660 PRINT #2,A$;: IF LOC(1)>0 THEN 570
670 'if transmission suspended by XOFF,
680 '   resume by sending XON
690 IF PAUSE THEN PAUSE=FALSE: PRINT #1,XON$;
700 'check for keyboard input again
710 GOTO 510
8999 'if error, display error number and retry
9000 PRINT "ERROR NO.";ERR: RESUME
```

Notes on the Program

- "Asynchronous" communication implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to communications file or to screen) are terminated with a semicolon (;). This stops the carriage return normally issued at the end of the list of values to be printed.

- Line 90, where all numeric variables are defined as integer, is coded because any program looking for speed optimization should use integer counters in loops where possible.

- Note in line 510 that INKEY$ will return a null string if no character is pending.

Operation of Control Signals

This section contains more detailed technical information that you may need to know to communicate with another computer or peripheral from BASIC.

The output from the Asynchronous Communications Adapter conforms to the EIA RS232-C standard for interface between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE). This standard defines a number of control signals that are transmitted or received by your IBM Personal Computer to control the interchange of data with another computer or peripheral. These signals are DC voltages that are either ON (greater than +3 volts) or OFF (less than -3 volts). See the *IBM Personal Computer Technical Reference* manual for details.

Control of Output Signals with OPEN

When you start BASIC on your IBM Personal Computer, the RTS (Request To Send) and DTR (Data Terminal Ready) lines are held OFF. When an OPEN "COM... statement is performed, both of these lines are normally turned ON. However, you can specify the **RS** option on the OPEN "COM... statement to suppress the RTS signal. The lines stay ON until the communications file is closed (by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the R option). Even if the OPEN "COM... statement fails with an error (as described below), the DTR line (and RTS line, if applicable) is turned ON and stays ON. This allows you to retry the OPEN without having to execute a CLOSE.

Use of Input Control Signals

Normally, if either the CTS (Clear To Send) or DSR (Data Set Ready) lines are OFF, then an OPEN "COM... statement will not execute. After one second, BASIC will return with a "Device Timeout" error (error code 24). The Carrier Detect (sometimes called Receive Line Signal Detect) can be either ON or OFF; it has no effect on the operation of the program.

However, you can specify how you want these lines tested with the **RS**, **CS**, **DS**, and **CD** options on the OPEN "COM... statement. Refer to "OPEN "COM... Statement" in Chapter 4 for details.

If any of the signals that are being tested are turned OFF while the program is executing, I/O statements associated with the communications file won't work. For example, when you execute a PRINT # statement after the CTS or DSR line is turned off, a "Device Fault" (code 25) or "Device Timeout" (code 24) error occurs. The RTS and DTR stay on even if such an error occurs.

You can test for a line disconnect by using the INP function to read the bits in the MODEM Status Register on the Asynchronous Communications Adapter. See the following section, "Testing for Modem Control Signals," for details.

Testing for Modem Control Signals

There are four input control signals picked up by the Asynchronous Communications Adapter. These signals are the CTS and DSR signals described previously, the Carrier Detect (sometimes called Received Line Signal Detect) (pin 8), and Ring Indicator (pin 22). You can specify how you want to test the CTS, DSR, and CD lines with the OPEN "COM... statement. Ring Indicator is not used at all by the communications function in BASIC.

If you need to test for any of these signals in a program, you can check the bits corresponding to these signals in the MODEM Status Register on the Asynchronous Communications Adapter. To read the eight bits in this register, you use the INP function—use INP(&H3FE) to read the register on an unmodified communications adapter, and INP(&H2FE) to read the register on a modified communications adapter. See the "Asynchronous Communications Adapter" section of the *Technical Reference* manual for a description of which bits in the Status Register correspond to which control signals. You can also use the Delta bits in this register to determine if transient signals have appeared on any of the control lines. Note that for a control signal to have meaning, the pin corresponding to that signal must be connected in the cable to your modem or to the other computer.

You can also test for bits in the Line Status Register on the Asynchronous Communications Adapter. Use INP(&H3FD) to access this register on an unmodified communications adapter, and INP(&H2FD) to access it on a modified communications adapter. Again, the bits are described in the *IBM Personal Computer Technical Reference* manual. These bits can be used to determine what types of errors have occurred on receipt of characters from the communications line or whether a break signal has been detected.

Direct Control of Output Control Signals

You can control the RTS or DTR control signals directly from a BASIC program with an OUT statement. The states (ON or OFF) of these signals are controlled by bits in the MODEM Control Register on the Asynchronous Communications Adapter. The address of this register is &H3FC on an unmodified communications adapter and &H2FC on a modified communications adapter. The *IBM Personal Computer Technical Reference* manual describes which of these bits correspond to which signals.

You can also modify bits in the Line Control Register on the Asynchronous Communications Adapter. You should be careful in modifying these bits as most of the bits in this register have been set by BASIC at the time an OPEN statement is executed and changing a bit could cause communications failure. The Line Control Register is at address &H3FB on an unmodified communications adapter and at address &H2FB on a modified communications adapter.

When changing bits in either the MODEM Control Register or the Line Control Register, you should first read the register (with an INP function) and then rewrite the register with only the pertinent bit or bits changed.

A bit you may wish to control in the Line Control Register is bit 6, the Set Break bit. This bit permits you to produce a Break signal on the communications send line. A Break is often used to signal a remote computer to stop transmission. Typically a Break lasts for half a second. To produce such a signal, you must turn ON the Set Break, wait for the desired time of the Break signal, and then turn the bit OFF. The following BASIC statements will produce a Break signal of approximately half a second duration on an unmodified communications adapter.

```
100 IC%=INP(&H3FB) 'get contents of modem register
110 IZ%=IC% OR &H40 'turn ON the Set Break bit
110 OUT &H3FB,IZ%  'transmit to modem control register
120 FOR I=1 TO 500: NEXT I 'delay half a second
130 OUT &H3FB,IC% 'turn Set Break bit OFF in register
```

Communication Errors

Errors occur on communication files in the following order:

1) When opening the file —
   a) "Device Timeout" if one of the signals to be tested (CTS, DSR, or CD) is missing.
2) When reading data —
   a) "Com buffer overflow" if overrun occurs.
   b) "Device I/O error" for overrun, break, parity, or framing errors.
   c) "Device Fault" if you lose DSR or CD.
3) When writing data —
   a) "Device Fault" if you lose CTS, DSR, or CD on a Modem Status Interrupt while BASIC was doing something else.
   b) "Device Timeout" if you lose CTS, DSR, or CD while waiting to put data in the output buffer.

Appendix G. ASCII Character Codes

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using PRINT CHR$(*n*), where *n* is the ASCII code. The column headed "Control Character" lists the standard interpretations of ASCII codes 0 to 31 (usually used for control functions or communications).

Each of these characters may be entered from the keyboard by pressing and holding the Alt key, then pressing the digits for the ASCII code on the numeric keypad. Note, however, that some of the codes have special meaning to the BASIC program editor—the program editor uses its own interpretation for the codes and may not display the special character listed here.

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	◘	BS	040	(
009	(tab)	HT	041	)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	‼	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

ASCII value	Character	ASCII value	Character
064	@	096	`
065	A	097	a
066	B	098	b
067	C	099	c
068	D	100	d
069	E	101	e
070	F	102	f
071	G	103	g
072	H	104	h
073	I	105	i
074	J	106	j
075	K	107	k
076	L	108	l
077	M	109	m
078	N	110	n
079	O	111	o
080	P	112	p
081	Q	113	q
082	R	114	r
083	S	115	s
084	T	116	t
085	U	117	u
086	V	118	v
087	W	119	w
088	X	120	x
089	Y	121	y
090	Z	122	z
091	[	123	{
092	\	124	¦
093	]	125	}
094	^	126	~
095	_	127	⌂

ASCII value	Character	ASCII value	Character
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	å	166	ª
135	ç	167	º
136	ê	168	¿
137	ë	169	⌐
138	è	170	¬
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	░
145	æ	177	▒
146	Æ	178	▓
147	ô	179	│
148	ö	180	┤
149	ò	181	╡
150	û	182	╢
151	ù	183	╖
152	ÿ	184	╕
153	Ö	185	╣
154	Ü	186	║
155	¢	187	╗
156	£	188	╝
157	¥	189	╜
158	Pt	190	╛
159	ƒ	191	┐

ASCII value	Character	ASCII value	Character
192	└	224	α
193	┴	225	β
194	┬	226	Γ
195	├	227	π
196	─	228	Σ
197	┼	229	σ
198	╞	230	μ
199	╟	231	τ
200	╚	232	Φ
201	╔	233	Θ
202	╩	234	Ω
203	╦	235	δ
204	╠	236	∞
205	═	237	Ø
206	╬	238	ϵ
207	╧	239	∩
208	╨	240	≡
209	╤	241	±
210	╥	242	≥
211	╙	243	≤
212	╘	244	⌠
213	╒	245	⌡
214	╓	246	÷
215	╫	247	≈
216	╪	248	°
217	┘	249	•
218	┌	250	·
219	█	251	√
220	▄	252	n
221	▌	253	^{2}
222	▐	254	■
223	▀	255	(blank ‘FF’)

Extended Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended code is returned by the INKEY$ system variable. A null character (ASCII code 000) will be returned as the first character of a two-character string. If a two-character string is received by INKEY$, then you should go back and examine the second character to determine the actual key pressed. Usually, but not always, this second code is the scan code of the primary key that was pressed. The ASCII codes (in decimal) for this second character, and the associated key(s) are listed on the following page.

Second Code	Meaning
3	(null character) NUL
15	(shift tab) |◄—
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	function keys F1 through F10 (when disabled as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Ins
83	Del
84-93	F11-F20 (Shift- F1 through F10)
94-103	F21-F30 (Ctrl- F1 through F10)
104-113	F31-F40 (Alt- F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Cursor Left (Previous Word)
116	Ctrl-Cursor Right (Next Word)
117	Ctrl-End
118	Ctrl-Pg Dn
119	Ctrl-Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,=
132	Ctrl-Pg Up

NOTES

Appendix H. Hexadecimal Conversion Table

Hex	Decimal	Hex	Decimal
1	1	10	16
2	2	20	32
3	3	30	48
4	4	40	64
5	5	50	80
6	6	60	96
7	7	70	112
8	8	80	128
9	9	90	144
A	10	A0	160
B	11	B0	176
C	12	C0	192
D	13	D0	208
E	14	E0	224
F	15	F0	240
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

NOTES

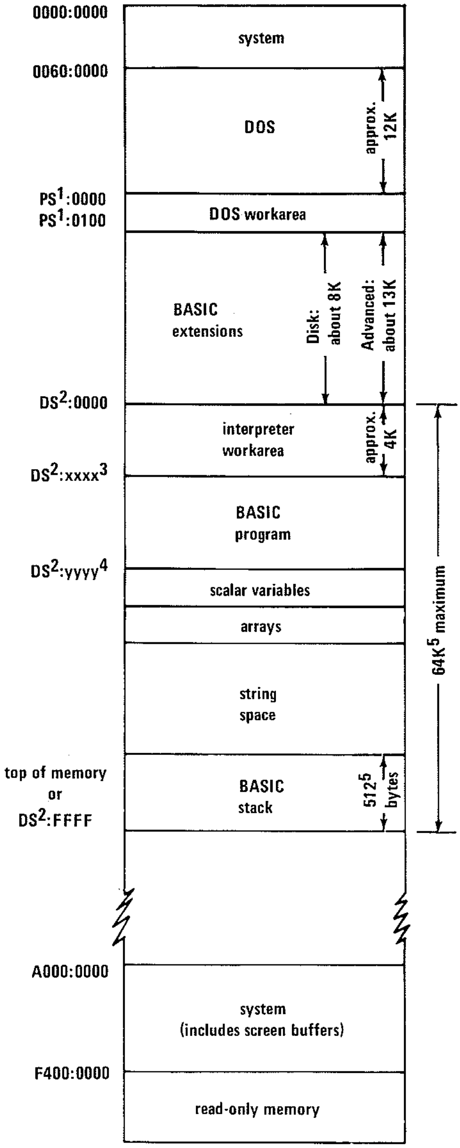
Appendix I. Technical Information and Tips

This appendix contains more specific technical information pertaining to BASIC. Included are a memory map, descriptions of how BASIC stores data internally, and some special techniques you can use to improve program performance.

Other information may be found in the *IBM Personal Computer Technical Reference* manual.

Memory Map

The following is a memory map for Disk and Advanced BASIC. DOS and the BASIC extensions are not present for Cassette BASIC. Addresses are in hexadecimal in the form *segment:offset.*

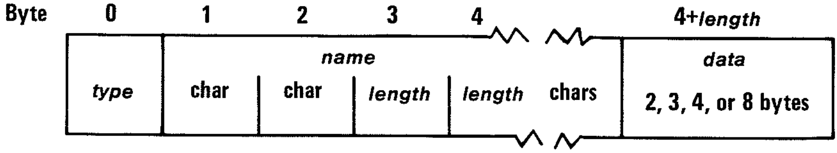

0000:0000
system
0060:0000
DOS
approx. 12K
PS1:0000
PS1:0100
DOS workarea
BASIC extensions
Disk: about 8K
Advanced: about 13K
DS2:0000
interpreter workarea
approx. 4K
DS2:xxxx3
BASIC program
DS2:yyyy4
scalar variables
arrays
string space
64K5 maximum
top of memory or DS2:FFFF
BASIC stack
5125 bytes
A000:0000
system (includes screen buffers)
F400:0000
read-only memory

Notes:

1. PS refers to DOS Program Segment
2. DS refers to BASIC's Data Segment
3. the number xxxx is in locations DS:30 and DS:31 (low byte, high byte)
4. the number yyyy is in locations DS:358, DS:359 (low byte, high byte)
5. or set by CLEAR command

How Variables Are Stored

Scalar variables are stored in BASIC's data area as follows:



type identifies the variable's type:

2	integer
3	string
4	single-precision
8	double-precision

name is the name of the variable. The first two characters of the name are stored in the bytes 1 and 2. Byte 3 tells how many more characters are in the variable name. These additional characters are stored starting at byte 4.

Note that this means any variable name will take up at least three bytes. A one- or two-character name will occupy exactly three bytes; an x character name will occupy x+1 bytes.

data follows the name of the variable, and may be either two, three, four, or eight bytes long (as described by *type*). The value returned by the VARPTR function points to this data.

For string variables, *data* is the *string descriptor:*

- The first byte of the string descriptor contains the length of the string (0 to 255).
- The last two bytes of the string descriptor contain the address of the string in BASIC's data space (the offset into the default segment). Addresses are stored with the low byte first and the high byte second, so:
  - The second byte of the string descriptor contains the low byte of the offset.
  - The third byte of the string descriptor contains the high byte of the offset.

For numeric variables *data* contains the actual value of the variable:

- Integer values are stored in two bytes, with the low byte first and the high byte second.
- Single-precision values are stored in four bytes in BASIC's internal floating point binary format.
- Double-precision values are stored in eight bytes in BASIC's internal floating point binary format.

BASIC File Control Block

When you call VARPTR with a file number as an argument, the returned value is the address of the BASIC file control block. The address is specified as an offset into BASICs Data Segment. (Note that the BASIC file control block is not the same as the DOS file control block.)

Information contained in the file control block is as follows (offsets are relative to the value returned by VARPTR):

Offset	Length	Description
0	1	The mode in which the file was opened: 1 - Input only 2 - Output only 4 - Random 16 - Append only 32 - Internal use 128 - Internal use
1	38	DOS file control block
39	2	For sequential files, the number of sectors read or written. For random files, contains 1 + the last record number read or written.
41	1	Number of bytes in sector when read or written.
42	1	Number of bytes left in input buffer.
43	3	(reserved)
46	1	Device number: 0,1 - Diskette drives A: and B: 248 - LPT3: 249 - LPT2: 250 - COM2: 251 - COM1: 252 - CAS1: 253 - LPT1: 254 - SCRN: 255 - KYBD:

Offset	Length	Description
47	1	Device width.
48	1	Position in buffer for PRINT #.
49	1	Internal use during LOAD and SAVE. Not used for data files.
50	1	Output position used during tab expansion.
51	128	Physical data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in sequential I/O mode.
179	2	Variable length record size. Default is 128. Set by length parameter on OPEN statement.
181	2	Current physical record number.
183	2	Current logical record number.
185	1	(reserved)
186	2	Diskette files only. Position for PRINT #, INPUT #, and WRITE #.
188	n	Actual FIELD data buffer. Size n is determined by the **/S:** option on the BASIC command. Use this offset to examine file data in random mode.

Keyboard Buffer

Characters typed on the keyboard are saved in the keyboard buffer until they are processed. Up to 15 characters can be held in the buffer; if you try to type more than 15 characters, the computer beeps.

INKEY$ will read only one character from the keyboard buffer even if there are several characters pending there. INPUT$ can be used to read multiple characters; however, if the requested number of characters are not already present in the buffer, BASIC will wait until enough characters are typed.

The system keyboard buffer may be cleared by the following lines of code:

```
DEF SEG=0: POKE 1050, PEEK(1052)
```

This technique could be useful, for example, to clear the buffer before you ask the user to "press any key."

BASIC has its own line buffer, where the program editor acts on characters that are received from the system keyboard buffer. BASIC's line buffer may be cleared using the following code:

```
DEF SEG: POKE 106,0
```

Search Order for Adapters

The printers associated with LPT1:, LPT2:, and LPT3: are assigned when you switch your computer on. The system looks for printer adapters in a particular sequence; the first printer adapter found becomes LPT1:, the second adapter (if one exists) becomes LPT2:, and the third (if it exists) becomes LPT3:. The search order is as follows:

1. An IBM Monochrome Display and Parallel Printer Adapter
2. A Parallel Printer Adapter
3. A Parallel Printer Adapter which has been modified to change its base address

If a printer was re-routed using the MODE command from DOS, the change is effective in BASIC as well.

The communication devices COM1: and COM2: are assigned in a manner similar to the printers. Their search order is:

1. An Asynchronous Communications Adapter
2. A modified Asynchronous Communications Adapter

Switching Displays

If you have both the Color/Graphics Monitor Adapter and the IBM Monochrome Display and Parallel Printer Adapter in your IBM Personal Computer, the one BASIC will normally write to would be the Monochrome Display. However, you can switch from one display to the other from BASIC by using the following code:

```
10 ' switch to monochrome adapter
20 DEF SEG = 0
30 POKE &H410, (PEEK(&H410) OR &H30)
40 SCREEN 0
50 WIDTH 40
60 WIDTH 80
70 LOCATE ,,1,12,13
```

```
10 ' switch to color adapter
20 DEF SEG = 0
30 POKE &H410, (PEEK(&H410) AND &HCF) OR &H10
40 SCREEN 1,0,0,0
50 SCREEN 0
60 WIDTH 40
70 LOCATE ,,1,6,7
```

Note: When you use this technique, the screen you are switching *to* is cleared. Also, you may need to keep track of the cursor location independently for each display.

Some Techniques with Color

Sixteen Background Colors: In text mode, if you are willing to give up blink, you can get all 16 colors (0-15) for the background color. Do the following:

In 40-column width: `OUT &H3D8,8`

In 80-column width: `OUT &H3D8,9`

Character Color in Graphics Mode: You can display regular text characters while in graphics mode. In medium resolution, the foreground color of the characters is color number 3; the background is color number 0.

You can change the foreground color of the characters from 3 to 2 or 1 by performing a:

```
DEF SEG: POKE &H4E, color
```

where *color* is the desired foreground color (1, 2, or 3— 0 is *not* allowed). Later PRINTs will use the specified foreground color.

Tips and Techniques

Often there are several different ways you can code something in BASIC and still get the same function. This section contains some general hints for coding to improve program performance.

GENERAL

- **Combine statements** where convenient to take advantage of the 255 character statement length. For example:

  Do

  ```
  100 FOR I=1 TO 10: READ A(I): NEXT I
  ```

  Instead of

  ```
  100 FOR I=1 TO 10
  110 READ A(I)
  120 NEXT I
  ```

- **Avoid repetitive evaluation of expressions.** If you do the identical calculation in several statements, you can evaluate the expression once and save the result in a variable for use in later statements. For example:

  Do

  ```
  300 X=C*3+D
  310 A=X+Y
  320 B=X+Z
  ```

  Instead of

  ```
  310 A=C*3+D+Y
  320 B=C*3+D+Z
  ```

  However, assigning a constant to a variable is faster than assigning the value of another variable to the variable.

- **Use simple arithmetic.** In general, addition is performed faster than multiplication, and multiplication is faster than division or exponentiation.

  Consider these example:

  ```
  Do                     Instead of

  250  B=A*.5            250  B=A/2
  500  B=A+A             500  B=A*2
  650  B=A*A*A*          650  B=A^3
  750  B%=A%\4           750  B%=INT(A%/4)
  ```

- **Use built-in functions.** Use the built-in system functions where possible; they always execute faster than the same capability written in BASIC.

- **Use remarks sparingly.** It takes a small amount of time for BASIC to identify a remark. Use the single quote (') to place remarks at the end of the line rather than using a separate statement for them when possible. This improves performance and saves storage by eliminating the need for a line number. For example:

  Do

  ```
  10  FOR I=1 TO 10
  20 A(I)=30  ' initialize A
  30 NEXT I
  ```

  Instead of

  ```
  10 FOR I=1 TO 10
  15 ' initialize A
  20 A(I)=30
  30 NEXT I
  ```

- Just a note about IBM Personal Computer BASIC— When BASIC wants to branch to a particular line number, it doesn't know exactly where in memory that line is. Therefore BASIC has to search through the line numbers in the program, starting at the beginning, to find the line it's looking for.

In some other BASICs, this search must be performed each time the branch occurs in the program. In IBM Personal Computer BASIC, the search is only performed once, and thereafter the branch is direct. So placing frequently-used subroutines at the beginning of the program will not make your program run faster.

LOGIC CONTROL

- **Use the capabilities of the IF statement.** By using AND and OR and the ELSE clause, you can often avoid the need for more IF statements and additional code in the program. For example:

  Do

  ```
  200 IF A=B AND C=D THEN Z=12 ELSE Z=B
  ```

  Instead of

  ```
  200 IF A=B THEN GOTO 210
  205 GOTO 215
  210 IF C=D THEN 225
  215 Z=B
  220 GOTO 230
  225 Z=12
  230 ...
  ```

- **Order IF statements** so the most frequently occurring condition is tested first. This avoids having to make extra tests. For example, suppose you have a data entry file for customer orders which consists of different record types and numerous individual transactions.

A typical record group looks like this:

Type code	Record type
A	Header
B	Customer name and address
C	Transaction
C	Transaction
.	.
.	.
C	Transaction
D	Trailer

Do

```
100 IF TYPE$="C" THEN 3000
110 IF TYPE$="A" THEN 1000
120 IF TYPE$="B" THEN 2000
130 IF TYPE$="D" THEN 4000
```

Instead of

```
100 IF TYPE$="A" THEN 1000
110 IF TYPE$="B" THEN 2000
120 IF TYPE$="C" THEN 3000
130 IF TYPE$="D" THEN 4000
```

If you had 100 groups, with 10 transactions per group, moving the test to the beginning of the list results in 1800 fewer IF statements being executed.

Another example of ordering IF statements in a cascade so less tests need to be performed:

DO

```
200 IF A<>1 THEN 250
210 IF B=1 THEN X=0
220 IF B=2 THEN X=1
230 IF B=3 THEN X=2
240 GOTO 280
250 IF B=1 THEN X=3
260 IF B=2 THEN X=4
270 IF B=3 THEN X=5
280 ...
```

Instead of

```
200 IF A=1 AND B=1 THEN X=0
210 IF A=1 AND B=2 THEN X=1
220 IF A=1 AND B=3 THEN X=2
230 IF A<>1 AND B=1 THEN X=3
240 IF A<>1 AND B=2 THEN X=4
250 IF A<>1 AND B=3 THEN X=5
```

LOOPS

- **Use integer counters** on FOR...NEXT loops when possible. Integer arithmetic is performed faster than single- and double-precision arithmetic.

- **Omit the variable on the NEXT statement** where possible. If you include the variable, BASIC takes a little time to check to see that it is correct. It may be necessary to include the variable on the NEXT statement if you are branching out of nested loops. Refer to "FOR and NEXT Statements" in Chapter 4 for more information.

- **Use FOR...NEXT loops** instead of using the IF, GOTO combination of statements.

For example:

Do

```
200 FOR I=1 TO 10
  .
  .
  .
  .
300 NEXT I
```

Instead of

```
200 I=1
210 ...
  .
  .
290 I=I+1
300 IF I<=10 THEN 210
```

- **Remove unnecessary code from loops.** This includes statements which don't affect the loop, as well as non-executable statements such as REM and DATA. For example:

Do

```
10 A=B+1
20 FOR X=1 TO 100
30 IF D(X)>A THEN D(X)=A
40 NEXT X
```

Instead of

```
10 FOR X=1 TO 100
20 A=B+1
30 IF D(X)>A THEN D(X)=A
40 NEXT X
```

In the preceding example, it is not necessary to calculate the value of A each time through the loop, because the loop never changes the value of A.

The next example shows a non-executable statement.

Do

```
200 DATA 5, 12, 1943
210 FOR I=1 TO 100
    .
    .
    .
300 NEXT I
```

Instead of

```
200 FOR I=1 TO 100
210 DATA 5, 12, 1943
    .
    .
    .
300 NEXT I
```

Refer also to “Performance Hints” in Appendix B for some tips relating to diskette files.

Appendix J. Glossary

This part of the book explains many of the technical terms you may run across while programming in BASIC.

absolute coordinate form: In graphics, specifying the location of a point with respect to the origin of the coordinate system.

access mode: A technique used to obtain a specific logical record from, or put a logical record into, a file.

accuracy: The quality of being free from error. On a machine this is actually measured, and refers to the size of the error between the actual number and its value as stored in the machine.

active page: On the Color/Graphics Monitor Adapter, the screen buffer which has information written to it. It may be different from the screen buffer whose information is being displayed.

adapter: A mechanism for attaching parts.

address: The location of a register, a particular part of memory, or some other data source or destination. Or, to refer to a device or a data item by its address.

addressable point: In computer graphics, any point in a display space that can be addressed. Such points are finite in number and form a discrete grid over the display space.

algorithm: A finite set of well-defined rules for the solution of a problem in a finite number of steps.

allocate: To assign a resource, such as a diskette file or a part of memory, to a specific task.

alphabetic character: A letter of the alphabet.

alphameric or alphanumeric: Pertaining to a character set that contains letters and digits.

application program: A program written by or for a user which applies to the user's work. For example, a payroll application program.

argument: A value that is passed from a calling program to a function.

arithmetic overflow: Same as overflow.

array: An arrangement of elements in one or more dimensions.

ASCII: American National Standard Code for Information Interchange. The standard code used for exchanging information among data processing systems and associated equipment. An ASCII file is a text file where the characters are represented in ASCII codes.

asynchronous: Without regular time relationship; unpredictable with respect to the execution of a program's instructions.

attribute: A property or characteristic of one or more items.

background: The area which surrounds the subject. In particular, the part of the display screen surrounding a character.

backup: Pertaining to a system, device, file, or facility that can be used in case of a malfunction or loss of data.

baud: A unit of signalling speed equal to the number of discrete conditions or signal events per second.

binary: Pertaining to a condition that has two possible values or states. Also, refers to the Base 2 numbering system.

bit: A binary digit.

blank: A part of a data medium in which no characters are recorded. Also, the space character.

blinking: An intentional regular change in the intensity of a character on the screen.

boolean value: A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is zero).

bootstrap: An existing version, perhaps a primitive version, of a computer program that is used to establish another version of the program. Can be thought of as a program which loads itself.

bps: Bits per second.

bubble sort: A technique for sorting a list of items into sequence. Pairs of items are examined, and exchanged if they are out of sequence. This process is repeated until the list is sorted.

buffer: An area of storage which is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read or from which data is written.

bug: An error in a program.

byte: The representation of a character in binary. Eight bits.

call: To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

carriage return character (CR): A character that causes the print or display position to move to the first position on the same line.

channel: A path along which signals can be sent, for example, a data channel or an output channel.

character: A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A connected sequence of characters is called a character *string*.

clock: A device that generates periodic signals used for synchronization. Each signal is called a clock pulse or clock tick.

code: To represent data or a computer program in a symbolic form that can be accepted by a computer; to write a routine. Also, loosely, one or more computer programs, or part of a program.

comment: A statement used to document a program. Comments include information that may be helpful in running the program or reviewing the output listing.

communication: The transmission and reception of data.

complement: An "opposite." In particular, a number that can be derived from a given number by subtracting it from another given number.

compression: Arranging data so it takes up a minimal amount of space.

concatenation: The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.

constant: A fixed value or data item.

control character: A character whose occurrence in a particular context initiates, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example, carriage return, font change, or end of transmission.

coordinates: Numbers which identify a location on the display.

cursor: A movable marker that is used to indicate a position on the display.

debug: To find and eliminate mistakes in a program.

default: A value or option that is assumed when none is specified.

delimiter: A character that groups or separates words or values in a line of input.

diagnostic: Pertaining to the detection and isolation of a malfunction or mistake.

directory: A table of identifiers and references to the corresponding items of data. For example, the directory for a diskette contains the names of files on the diskette (identifiers), along with information that tells DOS where to find the file on the diskette.

disabled: A state that prevents the occurrence of certain types of interruptions.

DOS: Disk Operating System. In this book, refers only to the IBM Personal Computer Disk Operating System.

dummy: Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.

duplex: In data communication, pertaining to a simultaneous two-way independent transmission in both directions. Same as full duplex.

dynamic: Occurring at the time of execution.

echo: To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit: To enter, modify, or delete data.

element: A member of a set; in particular, an item in an array.

enabled: A state of the processing unit that allows certain types of interruptions.

end of file (EOF): A "marker" immediately following the last record of a file, signalling the end of that file.

event: An occurrence or happening; in IBM Personal Computer Advanced BASIC, refers particularly to the events tested by the COM(n), KEY(n), PEN, and STRIG(n).

execute: To perform an instruction or a computer program.

extent: A continuous space on a diskette, occupied or reserved for a particular file.

fault: An accidental condition that causes a device to fail to perform in a required manner.

field: In a record, a specific area used for a particular category of data.

file: A set of related records treated as a unit.

fixed-length: Referring to something in which the length does not change. For example, random files have fixed-length records; that is, each record has the same length as all the other records in the file.

flag: Any of various types of indicators used for identification, for example, a character that signals the occurrence of some condition.

floppy disk: A diskette.

folding: A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.

font: A family or assortment of characters of a particular size and style.

foreground: The part of the display area that is the character itself.

format: The particular arrangement or layout of data on a data medium, such as the screen or a diskette.

form feed (FF): A character that causes the print or display position to move to the next page.

function: A procedure which returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing, or its characteristic action.

function key: One of the ten keys labeled F1 through F10 on the left side of the keyboard.

garbage collection: Synonym for housecleaning.

graphic: A symbol produced by a process such as handwriting, printing, or drawing.

half duplex: In data communication, pertaining to an alternate, one way at a time, independent transmission.

hard copy: A printed copy of machine output in a visually readable form.

header record: A record containing common, constant, or identifying information for a group of records that follows.

hertz (Hz): A unit of frequency equal to one cycle per second.

hierarchy: A structure having several levels, arranged in a tree-like form. "Hierarchy of operations" refers to the relative priority assigned to arithmetic or logical operations which must be performed.

host: The primary or controlling computer in a multiple computer installation.

housecleaning: When BASIC compresses string space by collecting all of its useful data and frees up unused areas of memory that were once used for strings.

implicit declaration: The establishment of a dimension for an array without it having been explicitly declared in a DIM statement.

increment: A value used to alter a counter.

initialize: To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

instruction: In a programming language, any meaningful expression that specifies one operation and its operands, if any.

integer: One of the numbers 0, ±1, ±2, ±3, ...

integrity: Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.

interface: A shared boundary.

interpret: To translate and execute each source language statement of a computer program before translating and executing the next statement.

interrupt: To stop a process in such a way that it can be resumed.

invoke: To activate a procedure at one of its entry points.

joystick: A lever that can pivot in all directions and is used as a locator device.

justify: To align characters horizontally or vertically to fit the positioning constraints of a required format.

K: When referring to memory capacity, two to the tenth power or 1024 in decimal notation.

keyword: One of the predefined words of a programming language; a reserved word.

leading: The first part of something. For example, you might refer to leading zeroes or leading blanks in a character string.

light pen: A light sensitive device that is used to select a location on the display by pointing it at the screen.

line: When referring to text on a screen or printer, one or more characters output before a return to the first print or display position. When referring to input, a string of characters accepted by the system as a single block of input; for example, all characters entered before you press the Enter key. In graphics, a series of points drawn on the screen to form a straight line. In data communications, any physical medium, such as a wire or microwave beam, that is used to transmit data.

line feed (LF): A character that causes the print or display position to move to the corresponding position on the next line.

literal: An explicit representation of a value, especially a string value; a constant.

location: Any place in which data may be stored.

loop: A set of instructions that may be executed repeatedly while a certain condition is true.

M: Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.

machine infinity: The largest number that can be represented in a computer's internal format.

mantissa: For a number expressed in floating point notation, the numeral that is not the exponent.

mask: A pattern of characters that is used to control the retention or elimination of another pattern of characters.

matrix: An array with two or more dimensions.

matrix printer: A printer in which each character is represented by a pattern of dots.

menu: A list of available operations. You select which operation you want from the list.

minifloppy: A 5-1/4 inch diskette.

nest: To incorporate a structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.

notation: A set of symbols, and the rules for their use, for the representation of data.

null: Empty, having no meaning. In particular, a string with no characters in it.

octal: Pertaining to a Base 8 number system.

offset: The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.

on-condition: An occurrence that could cause a program interruption. It may be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.

operand: That which is operated upon.

operating system: Software that controls the execution of programs; often used to refer to DOS.

operation: A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.

overflow: When the result of an operation exceeds the capacity of the intended unit of storage.

overlay: To use the same areas of memory for different parts of a computer program at different times.

overwrite: To record into an area of storage so as to destroy the data that was previously stored there.

pad: To fill a block with dummy data, usually zeros or blanks.

page: Part of the screen buffer that can be displayed and/or written on independently.

parameter: A name in a procedure that is used to refer to an argument passed to that procedure.

parity check: A technique for testing transmitted data. Typically, a binary digit is appended to a group of binary digits to make the sum of all the digits either always even (even parity) or always odd (odd party).

pixel: A graphics "point." Also, the bits which contain the information for that point.

port: An access point for data entry or exit.

position: In a string, each location that may be occupied by a character and that may be identified by a number.

precision: A measure of the ability to distinguish between nearly equal values.

prompt: A question the computer asks when it needs you to supply information.

protect: To restrict access to or use of all, or part of, a data processing system.

queue: A line or list of items waiting for service; the first item that went in the queue is the first item to be serviced.

random access memory: Storage in which you can read and write to any desired location. Sometimes called direct access storage.

range: The set of values that a quantity or function may take.

raster scan: A technique of generating a display image by a line-by-line sweep across the entire display screen. This is the way pictures are created on a television screen.

read-only: A type of access to data that allows it to be read but not modified.

record: A collection of related information, treated as a unit. For example, in stock control, each invoice might be one record.

recursive: Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.

relative coordinates: In graphics, values that identify the location of a point by specifying displacements from some other point.

reserved word: A word that is defined in BASIC for a special purpose, and that you cannot use as a variable name.

resolution: In computer graphics, a measure of the sharpness of an image, expressed as the number of lines per unit of length discernible in that area.

routine: Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

row: A horizontal arrangement of characters or other expressions.

scalar: A value or variable that is not an array.

scale: To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scan: To examine sequentially, part by part. See raster scan.

scroll: To move all or part of the display image vertically or horizontally so that new data appears at one edge as old data disappears at the opposite edge.

segment: A particular 64K-byte area of memory.

sequential access: An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.

stack: A method of temporarily storing data so that the last item stored is the first item to be processed.

statement: A meaningful expression that may describe or specify operations and is complete in the context of the BASIC programming language.

stop bit: A signal following a character or block that prepares the receiving device to receive the next character or block.

storage: A device, or part of a device, that can retain data. Memory.

string: A sequence of characters.

subscript: A number that identifies the position of an element in an array.

syntax: The rules governing the structure of a language.

table: An arrangement of data in rows and columns.

target: In an assignment statement, the variable whose value is being set.

telecommunication: Synonym for data communication.

terminal: A device, usually equipped with a keyboard and display, capable of sending and receiving information.

toggle: Pertaining to anything having two stable states; to switch back and forth between the two states.

trailing: Located at the end of a string or number. For example, the number 1000 has three trailing zeros.

trap: A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

truncate: To remove the ending elements from a string.

two's complement: A form for representing negative numbers in the binary number system.

typematic key: A key that repeats as long as you hold it down.

update: To modify, usually a master file, with current information.

variable: A quantity that can assume any of a given set of values.

variable-length record: A record having a length independent of the length of other records in the file.

vector: In graphics, a directed line segment. More generally, an ordered set of numbers, and so, a one-dimensional array.

wraparound: The technique for displaying items whose coordinates lie outside the display area.

write: To record data in a storage device or on a data medium.

INDEX

Special Characters

! 3-14
$ 3-13
% 3-14
?Redo from start 4-123
3-14

A

A: 3-35
ABS 4-23
absolute form for specifying coordinates 3-43
absolute value 4-23
accuracy 3-11
adapters
  communications I-8
  display 3-38, 4-31, 4-205, I-8
  printer I-7
adding characters 2-34
adding program lines 2-36
addition 3-21
Advanced BASIC 1-6
alphabetic characters 3-4
Alt 2-13
Alt-Ctrl-Del 2-18
Alt-key words 2-14
alternate shifts 2-13
AND 3-25
append 4-189, B-8
arctangent 4-25
arithmetic operators 3-21
arrays 3-15, 4-77, 4-87, 4-200
ASC 4-24
ASCII codes 4-24, 4-38, Appendix G
ASCII format 4-253
aspect ratio 4-42, 4-80
assembly language subroutines See machine language subroutines
assignment statement 4-139
ATN 4-25
AUTO 3-3, 4-26
automatic line numbers 4-26

B

B: 3-35
background 3-40, 4-49
Backspace 2-12, 2-28, 2-34
BASIC command 2-4
BASIC versions 1-3
BASIC, starting 2-3
BASIC's Data Segment 4-71
BEEP 4-28
beeping from the computer I-7
blanks 3-6, D-7
blinking characters 4-50, 4-51
BLOAD 4-29, C-5
Boolean operations 3-25
border screen 3-40, 4-49
branching 4-113, 4-180
Break 2-17, 2-29
bringing up BASIC 2-3
BSAVE 4-32

buffer
  communications 2-5, 4-195
  keyboard I-7
    reading the 4-119, 4-127
  random file 2-5, 4-106, 4-163, 4-230
  screen 3-41, 4-31, 4-33
built-in functions
  See functions

C

CALL 4-34, C-10
cancelling a line 2-35
capital letters 2-12
Caps Lock 2-12
Cassette BASIC 1-4
cassette I/O B-1
cassette motor 4-172
CAS1: 3-35, 4-151
CDBL 4-35
CHAIN 4-36, 4-57
changing BASIC program 2-36
changing characters 2-32
changing line numbers 2-39
changing lines anywhere on the screen 2-38
changing program lines 2-37
character color I-9
character set 3-4, Appendix G
CHR$ 4-38, G-1
CINT 4-40
CIRCLE 4-41
CLEAR 4-44
clear screen 4-48
clearing memory 2-38, 4-174
clearing the keyboard buffer I-7
clock 3-44, 4-262
CLOSE 4-46
CLS 4-48
COLOR 4-49
  in graphics modes 3-42, 4-54
  in text mode 3-40, 4-49
COM 4-56
command level 2-7, 2-32
commands 4-6
comments 3-4, 4-240
COMMON 4-36, 4-57
communications 4-194, Appendix F
  buffer size 2-5
  trapping 4-56, 4-176
comparisons
  numeric 3-23
  string 3-24
complement, logical 3-25
complement, two's 3-27, 3-28
computed GOSUB/GOTO 4-180
COM1: 3-35, I-8
COM2: 3-35, I-8
concatenation 3-31
conjunction 3-25
constants 3-9
CONT 4-58
control block, file I-5
converting
  character to ASCII code 4-24
  degrees to radians 4-60
  from number to string 4-272
  from numbers for random files 4-170
  from numeric to octal 4-175

  hexadecimal 4-115, H-1
  numbers from random files 4-63
  one numeric precision to another 3-18
  radians to degrees 4-25
  string to numeric 4-285
converting programs to IBM Personal Computer BASIC Appendix D
coordinates 3-43
copy display 2-13
correcting current line 2-32
COS 4-60
cosine 4-60
CSNG 4-61
CSRLIN 4-62
Ctrl 2-13
Ctrl-Break 2-17, 2-29
Ctrl-Num Lock 2-17
cursor 2-19
cursor control keys 2-19
Cursor Down key 2-21
Cursor Left key 2-22
cursor position 4-62, 4-155, 4-215
Cursor Right key 2-22
Cursor Up key 2-21
CVI, CVS, CVD 4-63, B-11

D

DATA 4-64, 4-238
Data Segment 4-71
DATE$ 4-66
DEBUG C-7
decisions 4-116
declaring arrays 3-15, 4-77
declaring variable types 3-13, 3-14, 4-73
DEF FN 4-68
DEF SEG 4-71
DEF USR 4-75
DEFtype (-INT, -SNG, -DBL, -STR) 3-14, 4-73
Del key 2-27
DELETE 3-3, 4-76
deleting a file 4-136
deleting a program 2-38, 4-174
deleting arrays 4-87
deleting characters 2-33
deleting program lines 2-37, 4-76
delimiting reserved words 3-6
descriptor, string I-4
device name 3-34, 3-35
Device Timeout 4-162, A-11
DIM 4-77
dimensioning arrays 3-15, 4-77
DIR 4-97
direct mode 2-7, 4-178
disjunction 3-25
diskette I/O Appendix B
display adapters 3-38, 4-31, 4-205, I-8
display pages 3-41, 4-258
display program lines 4-147
display screen, using 3-38
division 3-21
division by zero A-8
double-precision 3-11, 4-35
DRAW 4-79
DS (BASIC's Data Segment) 4-71
duplicating a program line 2-38

E

EDIT 3-3, 4-84
editor 2-19
editor keys 2-19
  Backspace 2-28
  Ctrl-Break 2-29
  Ctrl-End 2-25
  Ctrl-Home 2-20
  Cursor Down 2-21
  Cursor Left 2-22
  Cursor Right 2-22
  Cursor Up 2-21
  Del 2-27
  End 2-25
  Esc 2-28
  Home 2-20
  Ins 2-26
  Next Word 2-23
  Previous Word 2-24
  Tab 2-30
ELSE 4-116
END 4-85
End key 2-25
end of file 4-86, B-7
ending BASIC 4-278
Enter key 2-11
entering BASIC program 2-36
entering data 2-19
EOF 4-86, B-7
equivalence 3-25
EQV 3-25
ERASE 4-87
ERASE (DOS) 4-136
erasing a file 4-136
erasing a program 2-38, 4-174
erasing arrays 4-87
erasing characters 2-33
erasing part of a line 2-35
erasing program lines 2-37, 4-76
erasing variables 4-44
ERL 4-89
ERR 4-89
ERROR 4-91
error codes 4-89, 4-91, Appendix A
error line 4-89
error messages Appendix A
error trapping 4-89, 4-91, 4-178, 4-245
Esc key 2-28
event trapping
  COM(n) (communications activity) 4-56, 4-176
  KEY(n) 4-134, 4-182
  PEN 4-185, 4-206
  STRIG(n) (joystick button) 4-187, 4-275
exchanging variables 4-277
exclusive or 3-25
executable statements 3-3
executing a program 2-4, 4-251
EXP 4-93
exponential function 4-93
exponentiation 3-21
expressions
  numeric 3-21
  string 3-31
extended ASCII codes G-6
extension, filename 3-36

F

false 3-23, 3-25
FIELD 4-94
file control block I-5
file specification 3-34
filename 3-34, 3-36
filename extension 3-36

files 3-33, Appendix B, D-1
  control block I-4
  file number 3-33
  maximum number 2-4
  naming 3-34
  opening 3-33, 4-189
  position of 4-153
  size 4-158
FILES 4-97
FIX 4-99
fixed point 3-9
fixed-length strings 4-163
floating point 3-9
floor function 4-130
flushing the keyboard buffer I-7
folding, line 2-27
FOR 4-100, I-14
foreground 3-40, 4-49
format notation v
formatting 4-219
FRE 4-104
free space 2-5, 4-44, 4-104
frequency table 4-263
function keys 2-9
functions 3-29, 3-32, 4-5, 4-17, I-11
  derived Appendix E
  user-defined 4-68

G

garbage collection 4-104
GET (files) 4-106, B-10
GET (graphics) 4-108
glissando 4-264
GOSUB 4-111, 4-180
GOTO 4-113, 4-180
graphics 3-38, D-1
graphics modes 3-41, 4-257
graphics statements
  CIRCLE 4-41
  COLOR 4-54
  DRAW 4-79
  GET 4-108
  LINE 4-141
  PAINT 4-203
  POINT function 4-213
  PSET and PRESET 4-228
  PUT 4-232

H

hard copy of screen 2-13
HEX$ 4-115
hexadecimal 3-10, 4-115, H-1
hierarchy of operations 3-29
high resolution 3-43, 4-257
high-intensity characters 4-50, 4-51
hold 2-17
Home key 2-20
housecleaning 4-104

I

I/O statements 4-13, Appendix B
IF 4-116, D-2, I-12
IMP 3-25
implication 3-25
implicit declaration of arrays 3-17
index (position in string) 4-129
indirect mode 2-7
initializing BASIC 2-3
INKEY$ 4-119, G-6
INP 4-121
INPUT 4-122

INPUT# 4-125
input and output 3-33
input file mode 4-189, B-5
INPUT$ 4-127, F-3
Ins key 2-26
insert mode 2-26
inserting characters 2-34
INSTR 4-129
INT 4-130
integer 3-9, 3-11
  converting to 4-40, 4-99, 4-130
integer division 3-22
interrupting program execution 2-17
intrinsic functions
  See functions
invisible characters 4-51

J

joystick 3-45, 4-268
joystick button 4-187, 4-273, 4-275
jumping 4-113, 4-180

K

KEY 4-131
KEY(n) 4-134
keyboard 2-8
  buffer See buffer, keyboard
  input 4-119, 4-122, 4-127, 4-144
KILL 4-136, B-3
KYBD: 3-35

L

last point referenced 3-43
LEFT$ 4-137
left-justify 4-163
LEN 4-138
length of file 4-158
length of string 4-104, 4-138
LET 4-139
light pen 3-45, 4-185, 4-206
LINE 4-141
line feed 2-32, 4-191, D-3
LINE INPUT 4-144
LINE INPUT# 4-145
lines
  BASIC program 3-3
  drawing in graphics 4-141
  folding 2-27
  line numbers 2-7, 3-3, 4-26, 4-241
  on screen 3-39
LIST 3-3, 4-147
list program lines 4-149
listing files
  on cassette 4-151
  on diskette 4-97
LLIST 4-149
LOAD 4-150, B-2
loading binary data 4-29
LOC 4-153
LOCATE 4-155
LOF 4-158
LOG 4-159
logarithm 4-159
logical line 2-32
logical operators 3-25, D-3
loops 4-100, 4-292, I-14
LPOS 4-160
LPRINT 4-161
LPRINT USING 4-161

LPT1: 3-35, 4-149, 4-160, 4-161, I-7
LPT2: 3-35, I-7
LPT3: 3-35, I-7
LSET 4-163

M

machine language subroutines 4-34, 4-75, 4-284, Appendix C
medium resolution 3-42, 4-257
memory image 4-32
memory map I-2
MERGE 4-36, 4-165, B-3
messages Appendix A
MID$ 4-167, D-6
MKI$, MKS$, MKD$ 4-170, B-9
MOD 3-22
modulo arithmetic 3-22
MOTOR 4-172
multiple statements on a line 3-3
multiplication 3-21
music 3-44, 4-209

N

NAME 4-173
naming files 3-34
negation 3-21
NEW 4-174
NEXT 4-100
  See also FOR
Next Word 2-23
non-executable statements 3-3
NOT 3-25
Num Lock 2-16
numeric characters 3-4
numeric comparisons 3-23
numeric constants 3-9
numeric expressions 3-21
numeric functions 3-29, 4-17
numeric keypad 2-15
numeric variables 3-13

O

OCT$ 4-175
octal 3-10, 4-175
Ok prompt 2-7
ON COM(n) 4-176
ON ERROR 4-178
ON KEY(n) 4-182
ON PEN 4-185
ON STRIG(n) 4-187
ON...GOSUB 4-180
ON...GOTO 4-180
OPEN (file) 4-189, B-4, B-9
OPEN "COM... 4-194, F-6
operators
  arithmetic 3-21
  concatenation 3-31
  functions 3-29, 3-32
  logical 3-25
  numeric 3-21
  relational 3-23
  string 3-31
OPTION BASE 4-200
options on BASIC command 2-4
OR 3-25
or, exclusive 3-25
order of execution 3-29
OUT 4-201

output file mode 4-189, B-4
overflow A-7
overlay 4-36
overscan 3-40

P

paddles 3-45
PAINT 4-203
palette 3-42, 4-54
parentheses 3-30
pause 2-17
PEEK 4-205, D-4
PEN 4-206
performance hints B-15, I-10
Pg Up and Pg Dn 2-16
PLAY 4-209
POINT 4-213
POKE 4-214, C-4, D-4
POS 4-215
position in string 4-129
position of file 4-153
positioning the cursor 4-155
precedence 3-29
precision 3-11, 4-73
PRESET 4-228
Previous Word 2-24
PRINT 4-216
PRINT# 4-225
PRINT# USING 4-225
print formatting 4-219
print screen 2-13
PRINT USING 4-219
printing 4-161
program editor 2-19
protected files 4-253, B-3
PrtSc 2-13
PSET 4-228
PUT (files) 4-230, B-9
PUT (graphics) 4-232

R

random files 4-94, 4-106, 4-189, B-8
random numbers 4-236, 4-249
RANDOMIZE 4-236
READ 4-64, 4-238
record length
  maximum 2-5
  setting 4-189
?Redo from start 4-123
related publications vi
relational operators 3-23
relative form for specifying coordinates 3-43
REM 4-240
remarks 3-4, 4-240
RENAME 4-173
renaming files 4-173, B-3
RENUM 4-36, 4-89, 4-241
repeating a string 4-276
replacing program lines 2-37
requirements See system requirements
reserved words 3-6, 3-13
RESET 4-243
RESTORE 4-244
RESUME 4-245
resume execution 4-58
RETURN 4-111, 4-247
reverse image characters 4-51
RIGHT$ 4-248
right-justify 4-163
RND 4-249
rounding 3-18, D-5
rounding to an integer 4-40
RSET 4-163
RS232 See communications
RUN 4-251, B-2

S

SAVE 4-253, B-2
saving binary data 4-32
screen 3-39
  shifting 4-201
  use of 3-38
SCREEN function 4-255
SCREEN statement 4-257
SCRN: 3-35
Scroll Lock 2-16
scrolling 3-40
search order for adapters I-7
seeding random number generator 4-236
segment of storage 4-71
sequential files 4-189, B-4
SGN 4-260
shifting screen image 4-201
sign of number 4-260
SIN 4-261
sine 4-261
single-precision 3-11, 4-61
soft keys 2-9, 4-131
SOUND 4-262
sounds 3-44, 4-28, 4-209, 4-262
SPACE$ 4-265
spaces 3-6, D-7
SPC 4-266
special characters 3-5
specification of files 3-34
specifying coordinates 3-43
SQR 4-267
square root 4-267
stack space 4-44
starting BASIC 2-3
statements 4-8
  I/O 4-13
  non-I/O 4-8
STICK 4-268
STOP 4-270
STR$ 4-272
STRIG 4-273
STRIG(n) 4-275
string comparisons 3-24
string constants 3-9
string descriptor I-4
string expressions 3-31
string functions 3-32, 4-21, D-6
string space 2-5, 4-44, 4-104
string variables 3-13
STRING$ 4-276
subroutines 4-111, 4-180, I-11
subroutines, machine language 4-34, 4-75, 4-284, Appendix C
subscripts 3-15, 4-77, 4-200
substring 4-137, 4-167, 4-248
subtraction 3-21
SWAP 4-277
switching displays I-8
syntax diagrams v
syntax errors 2-40
SYSTEM 4-278
system functions
  See functions
system requirements
  Advanced 1-6
  Cassette 1-4
  Disk 1-5
System Reset 2-18

T

TAB 4-279
Tab key 2-30
TAN 4-280
tangent 4-280
technical information I-1

telecommunications
  See communications
tempo table 4-264
terminating BASIC 4-278
text mode 3-39, 4-257
THEN 4-116
TIME$ 4-281
tips I-10
trace 4-283
trigonometric functions
  arctangent 4-25
  cosine 4-60
  sine 4-261
  tangent 4-280
TROFF 4-283
TRON 4-283
true 3-23, 3-25
truncation 4-99, 4-130
truncation of program lines 2-36
two's complement 3-27, 3-28
type declaration characters 3-14
typewriter keyboard 2-10

U

underflow A-7
underlined characters 4-51
uppershift 2-12
user workspace 2-5, 4-44, 4-104
user-defined functions 4-68
using the screen 3-38
USR 4-75, 4-284, C-14

V

VAL 4-285
variables 3-12
  names 3-12
  storage of I-3
VARPTR 4-286, I-3
versions 1-3
visual page
  See display pages

W

WAIT 4-290
WEND 4-292
WHILE 4-292
WIDTH 4-294
word 2-23
workspace 2-5, 4-44, 4-104
WRITE 4-298
WRITE # 4-299

X

XOR 3-25

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________
Address ______________________________
City ______________ State ______________
Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ______________________________

Address ______________________________

City ______________ State ______________

Zip Code ______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC 6025013

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the Authorized IBM Personal Computer Dealer in your area.

Comments:

If you wish a reply, provide your name and address in this space.

Name ____________________________________

Address __________________________________

City ____________________ State __________________

Zip Code _______________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape