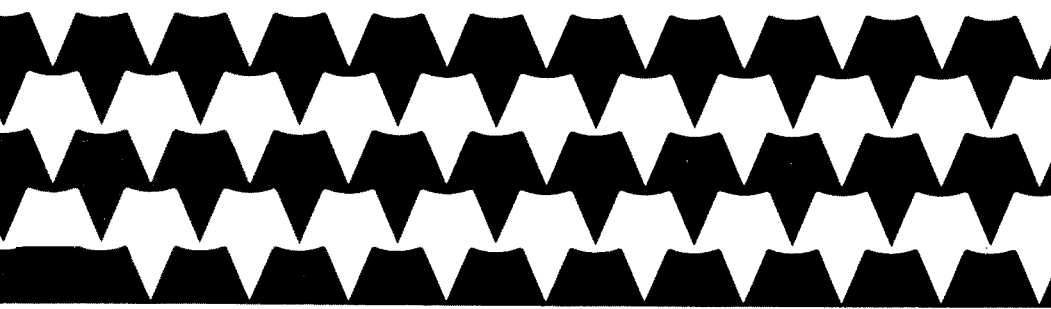


Tandy 3000

BASIC

Reference Manual



TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND
SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL
STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER's sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capacity, capability, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER's exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."** NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.
- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

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

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

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

TANDY 3000

BASIC REFERENCE MANUAL

GW™-BASIC Software: Copyright 1983, 1984, 1985 Microsoft Corporation. Licensed to Tandy Corporation. All Rights Reserved.

MS-DOS® Software: Copyright 1981, 1985 Microsoft Corporation. Licensed to Tandy Corporation. All Rights Reserved.

Compatibility Software Copyright 1985. Phoenix Software Associates Ltd. All Rights Reserved.

BASIC Reference Manual: Copyright 1985, 1986 Tandy Corporation. All Rights Reserved.

MS-DOS and Microsoft are registered trademarks of Microsoft Corporation.

GW is a trademark of Microsoft Corporation.

Tandy is a registered trademark of Tandy Corporation.

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

Contents

Introduction to BASIC	1
About this Manual	1
Notations	1
Terms	2
Chapter 1 / About BASIC for MS-DOS	3
Disk Files	3
Pathnames	4
Directory Paths	4
Names	5
Wildcards	5
Device Names	6
Chapter 2 / Loading BASIC	7
Loading BASIC via BASICA	7
Options for Loading BASIC	8
Redirection of Input and Output	11
Chapter 3 / Sample Session	13
Loading BASIC	13
Typing the Program	13
Saving the Program on Disk	14
Loading the Program into Memory	14
Chapter 4 / General Information	17
Editing	17
Sample Editing Session	18
Special Keys	20
The ALT Key	22
The PRT SC Key	22
Chapter 5 / Basic Concepts	23
Elements of a Program	23
Data	24
Constants	26
Variables	27
Declaring Numeric Constants and Variables	27
Numeric Constants	28
Numeric Variables	28

Numeric Precision Conversion	29
Manipulating Data	31
Arithmetic Operators	31
String Operator	32
Relational Operators	32
Logical Operators	34
Hierarchy of Operators	36
Functions	37
Chapter 6 / Arrays	39
Types of Arrays	42
Defining Arrays	43
Chapter 7 / Disk Files	45
Sequential Access Files	45
Creating a Sequential Access File	45
Updating a Sequential Access File	47
Direct Access Files	48
Creating a Direct Access File	49
Accessing a Direct Access File	50
Chapter 8 / Displaying Text and Graphics	53
Graphics Capability	53
Color	53
Colors in Mode 0	54
Colors in Mode 1	55
Colors in Mode 2	55
Resolution	56
Text Width	57
Video Memory	58
Summary	58
Specifying Coordinates	59
Chapter 9 / Introduction to BASIC Keywords ..	61
Format for Chapter 10	61
Terms Used in Chapter 10	62
Statements	63
Functions	68
Chapter 10 / BASIC Keywords	71

Chapter 11 / Technical Information	317
Interfacing With Assembly-Language Routines	317
Memory Allocation Outside BASIC's Work Area ...	317
Memory Allocation Inside BASIC's Work Area ...	317
Converting Subroutines	320
CALL Statement	321
CALLS Statement	323
USR Function	323
How Variables are Stored	325
Accessing String Variables	326
File Control Block	326
User Installed Devices	329
Information for Creating Child Processes	329
Chapter 12 / BASIC Error Codes and Messages	331
Appendix A / BASIC Reserved Words and Derived Functions	339
Appendix B / Keyboard and Character Code Charts	343
Keyboard ASCII/Scan Codes	343
ASCII Character Codes	346
Appendix C / Video Display Worksheet	353
Appendix D / Extended Codes	355
Index	357

INTRODUCTION TO BASIC

About This Manual

This manual describes BASIC for MS-DOS. It is a reference manual, not a tutorial. We assume you already know BASIC and are using this manual to locate information quickly. If you do not know BASIC, check your local bookstore for books for the novice programmer.

Notations

The following notations are used throughout this manual:

CAPITALS Material that you must enter exactly as it appears.

italics Items within command lines for which you must supply words, letters, characters, or values from a set of acceptable entries. Elsewhere, italics are used for technical terms.

... (ellipsis) Items preceding the ellipsis may be repeated.

[] Items enclosed in brackets are optional.

&Hnnnn nnnn is a hexadecimal number.

&Onnnnn nnnnn is an octal number.

keyname A key on your keyboard.

␣ A blank character (ASCII code 32). For example, in

BASIC␣␣PROG

two spaces are between BASIC and PROG.

Terms

The following terms are used in this manual:

buffer	An area in memory that BASIC uses to create and access a disk file. A buffer is represented by a number in the range 1 to 15. Once you use a buffer to create a file, you cannot use it to create or access any other files; you must first close the file. You may only access an open file with the buffer used to open it.
parameters	Information you supply to specify how a command is to operate.
arguments	Expressions you supply for a function to evaluate.
syntax	A command with its parameter(s), or a function with its argument(s). This shows the format to use for entering a keyword in a program line.

ABOUT BASIC FOR MS-DOS

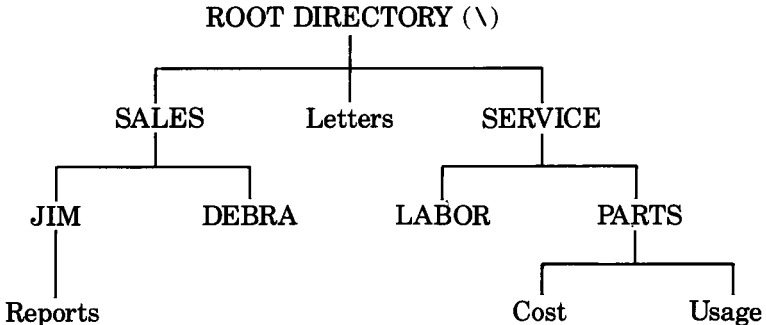
BASIC for MS-DOS® is an interpreter. This means that, when you run a program, BASIC looks at one statement at a time and executes it before going to the next statement.

BASIC also lets you take advantage of many MS-DOS features, such as:

- Multilevel directories
- Faster running programs
- Expanded graphics capabilities

Disk Files

BASIC uses the MS-DOS multilevel directory structure on disk. A formatted disk has a main directory called the *root* directory which is represented by the backslash (\). The root directory can contain both files and *second-level* directories.



This illustration demonstrates how a typical system for a sales and service company might be set up. The root directory contains two second-level directories: SALES and SERVICE. The root directory also contains a file called Letters.

Pathnames

BASIC lets you specify *pathnames* to access disk files just as MS-DOS does. A pathname is enclosed in quotation marks and may be a maximum of 63 characters. It contains the following information:

`"[d:][path]filename[.ext]"`

The drive is specified by *d*: and may be either A, B, C, or D. If you omit *d*:, BASIC uses the current drive.

Path gives the directory path for *filename*. The directory names are separated by a backslash (\). If you omit *path*, BASIC uses the current directory.

Filename specifies the name of the file being accessed. Filenames are 1 to 8 characters long. If a filename is more than 8 characters, BASIC truncates it to 8 characters.

Ext is an optional extension to the filename. Use extensions to help distinguish types of files. Extensions are always preceded by a period (.) and are 1 to 3 characters long. If you omit the extension, BASIC assumes .bas. Here are some common extensions:

.bas	for	BASIC programs
.txt	for	ASCII text
.dat	for	Data files
.obj	for	Object code
.rel	for	Relocatable code
.src	for	Source code

Some commands, such as CHDIR, MKDIR and RMDIR, require you to specify a directory instead of a file. In such cases, the pathname does not include a filename.

Names

Directory names and filenames must conform to MS-DOS conventions. They may contain any of the following characters:

- uppercase letters (A-Z)
- lowercase letters (a-z)
- decimal digits (0-9)

In addition, they may contain the following special characters:

\$ & # % () - @ { } ~ ` ! ' ^ _

An entire pathname may contain a maximum of 63 characters.

Some sample filenames are:

Prog.bas

Report

Telephon.sls

Some sample directory names are:

MEMOS

LETTERS

EMPLOYEE

In the command lines in this manual, directory names are in uppercase letters, and filenames are in lowercase letters. This lets you easily distinguish a directory name from a filename. (In copy, filenames begin with an uppercase letter so that they “jump out” from the other words on the page.) But, remember, you can type in whichever combination of uppercase and/or lowercase is easiest.

Wildcards

BASIC follows MS-DOS usage of wildcard notations when searching for directories or filenames. The wildcard notations are:

- ? indicates that any character can occupy that position.
- * indicates that any character can occupy that position or the remaining positions in the filename or extension.

For example, if you specify this filename:

Data?tst.txt

BASIC might find these files:

Data1tst.txt

Data3tst.txt

If you specify this filename:

`Data*.txt`

BASIC finds those files and might also find these:

`Data.txt`

`Datatst.txt`

`Data1.txt`

Device Names

BASIC uses device identifiers (dev:) to indicate a physical device to be used for communication. These names, which must be enclosed in quotation marks, are:

KYBD: keyboard. Use for input only.

SCRN: screen. Use for output only.

LPT1: printer. Use for output only.

COMn: RS232 Communications Channel 1 or 2. Use for input or output.

You can open any of these devices just as you would a disk file.

LOADING BASIC

Your MS-DOS/BASIC diskette contains both the BASIC Interpreter and the operating system required to run it. Before using BASIC, make a backup of this diskette by following the instructions in your *MS-DOS Handbook*. Then, start up MS-DOS from Drive A as described in the *Handbook*.

If your system contains Drive C (the hard disk system drive), you have the option of running MS-DOS and BASIC from that drive. Follow the instructions in the *Handbook* to copy the software to the drive, and restart your system under hard disk control.

When MS-DOS displays its system prompt (A> for floppy disk control or C> for hard disk control), you can load BASIC by typing one of two commands.

BASIC

immediately loads BASIC into the computer's memory.

BASICA

loads the small loader program BASICA.COM, which in turn loads BASIC. This alternate method of loading BASIC is discussed in detail in the section "Loading BASIC via BASICA."

Once you load BASIC, a paragraph of copyright information appears on your screen, followed by BASIC's prompt:

□k

At this point, you can begin using BASIC. The next chapter provides a sample session on loading BASIC and using some of its capabilities. If you don't want to use any of BASIC's other options for loading, go ahead to the next chapter.

Loading BASIC via BASICA

Some computers require you to type BASICA to load BASIC. To increase compatibility with such computers, your computer also accepts the BASICA command. When you enter the command, the computer executes the program BASICA.COM, which in turn loads BASIC.

In addition to compatibility, other advantages of loading BASIC via BASICA are as follows:

- BASIC is loaded at a different memory location than it would be otherwise. This feature lets you run a few BASIC programs that make use of certain memory locations that BASIC would otherwise have occupied.
- You can gain space on your program or system diskette because you can store the BASIC.EXE file on a separate disk.

The only limitations imposed by BASICA are:

- The /I option switch (discussed below) is always on.
- The communications buffer size is limited to 40K bytes if the system has 1 RS232 card or 20K bytes if it has 2 cards.

After you type BASICA , BASICA.COM searches the current directory for the file BASIC.EXE. If it finds BASIC.EXE, BASICA.COM loads it and passes control to it.

If BASICA.COM does not find BASIC.EXE, it asks you to replace your program disk with a disk that contains the file. Place a disk containing BASIC.EXE in any drive, and press . The program searches all drives, beginning with the current drive, until it finds BASIC.EXE or until you press .

After finding BASIC.EXE, the program asks you to re-insert your program disk if you removed it. Put the disk back in the drive, and press . The program transfers control to BASIC.

Options for Loading BASIC

When you load BASIC, you can also specify a set of options, which includes:

BASIC [*pathname*][<*input-file*] [>[>]*output-file*] [/F:# of files]
[/M:*highest memory location,maximum block size*[/C:*buffer size*]
[/S:*record length*[/D]][/I]

If you load BASIC by typing BASICA , the /I switch is always invoked. Other than that, you have the same options, regardless of how you load BASIC.

Pathname specifies a program to run immediately after BASIC is started.

<*Input-file* tells BASIC to receive input from *input-file* instead of the usual standard input (the keyboard). This option must follow *pathname* and precede all other options in the command line. Redirection of input and output is discussed later in this chapter.

>[>]*Output-file* redirects BASIC's output to *output-file* instead of the standard output (video display). If you specify 1 greater-than sign, *output file* is overwritten. If you use 2 greater-than signs, it is appended. This option must follow *input file* (if given) and precede all other options in the command line. Redirection of input and output is discussed later in this chapter.

/F: specifies the maximum number of data files that may be open at any one time. If you specify the /F: option, you must also specify the /I option. If you omit this option, the number of files defaults to three. The number of open files that MS-DOS supports depends on the value given for the FILES= command in the CONFIG.SYS file. We recommend that you set FILES=10 for BASIC. BASIC automatically reserves 4 files for internal use. This leaves 6 for BASIC file I/O; thus /F:6 is the maximum supported by MS-DOS when FILES= command is set to 10 in the CONFIG.SYS file.

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

/S: specifies the maximum record size for direct access files. If you use the /S: option, you also must specify the /I option. If you omit the /S: option, BASIC assumes 128 bytes.

/C: specifies the size of the receive buffer for each RS232 communications channel present in the system. The maximum amount you can specify depends on the number of RS232 cards present in the system and on the method used to load BASIC.

Loading Method	Number of Ports	Buffer Size
BASIC	1	64K bytes
BASIC	2	32K bytes
BASICA	1	40K bytes
BASICA	2	20K bytes

If you omit the /C: option, BASIC allocates 256 bytes for each receive buffer. The transmit buffer is always 128 bytes.

/M: sets the amount of memory space for BASIC to use by specifying the highest memory location available. Omit this option unless you plan to call assembly-language subroutines. BASIC can only allocate 64K bytes of memory. If you omit this option, the system allocates all 64K bytes of memory to BASIC.

If you plan to load assembly-language programs above BASIC's memory space, specify the optional *maximum block size* parameter to preserve space for both BASIC and your programs. Specify the value for *maximum block size* in blocks of 16 bytes. If you omit this parameter, 4096 blocks are used ($16 * 4096 = 65536$). This is the same amount reserved for BASIC; therefore, none is preserved for your routines. Specifying /M:32000,2048 allocates a maximum of 32768 bytes; BASIC uses the lower 32000 bytes. This leaves 768 bytes for your routines.

The *maximum block size* option is necessary if you plan to use the SHELL statement. If you do not preserve the memory space, COMMAND loads on top of your assembly-language routines when the SHELL statement executes.

/I tells BASIC not to dynamically allocate space during file operations. If you use the /F or /S switches, then you must specify /I. If you omit /I, BASIC dynamically allocates space. If you load BASIC via BASICA, /I is always invoked.

/D tells BASIC to load the Double Precision Transcendental math package into memory. This lets BASIC routines return double precision values. This package remains resident until you exit BASIC.

Examples

BASIC debits

initializes BASIC to 3 data files with all memory available. BASIC then loads and runs the program Debits.

BASIC payroll /F:5

initializes BASIC to 5 data files with all memory available. BASIC then loads and runs the program Payroll.

BASIC /M:21000

initializes BASIC to 3 data files and sets the highest memory location to be used by BASIC at 21000, the first 21000 bytes of BASIC's data segment.

BASIC budget /D

initializes BASIC to 3 data files with all memory available. BASIC loads and uses the Double Precision Transcendental math package.

Redirection of Input and Output

BASIC lets you redirect input and output. The syntax to redirection is:

BASIC [*pathname*] [<*input-file*] [>[>]*output-file*]

You can redirect standard input, normally from the keyboard, to the file *input-file*.

Standard output, normally to the video display, can be redirected to the file *output-file*. If *output-file* already exists, it is overwritten. You can, however, append the *output-file* to the existing file by using the append notation: >>*output-file*. If *output-file* does not exist, it is created.

The following BASIC statements use standard input:

INPUT
INPUT\$

LINE INPUT
INKEY\$

The BASIC statements **PRINT** and **WRITE** access standard output.

Error messages are sent both to the standard output and to the redirected output.

Examples

```
BASIC daily >daily.out
```

initializes BASIC and runs the program Daily. Redirects all output, normally sent to the screen, to the file Daily.out.

```
BASIC daily <daily.in
```

initializes BASIC and executes the program Daily. Daily receives all input, normally entered through the keyboard from the file Daily.in.

```
BASIC sample <tstdata.in >tstdata.out
```

initializes BASIC and executes the program Sample. Sample receives input from Tstdata.in and sends output to Tstdata.out.

```
BASIC payroll <week25 >>ytdtotal
```

initializes BASIC and executes the program Payroll. Payroll receives input from the file Week25. The output is appended to the file Ytdtotal.

Hints for redirection of input and output:

- File input from the KYBD: device still reads from the keyboard.
- File output to the SCRNL: device still outputs to the screen.
- BASIC still traps keys when you use the ON KEY() statement.
- **CTRL BREAK** tells BASIC to close all open files, then issue the message Break in line xxxx to standard output. Control returns to MS-DOS.
- Redirected input continues until BASIC receives a **CTRL Z**. This condition can be tested by the EOF() function. If the input file is not terminated by a **CTRL Z** or a BASIC file input statement tries to read past the end-of-file, BASIC closes any open files and issues the message Read past end to standard output. Control returns to MS-DOS.
- The printer echo key combination (**CTRL PRT SC**), which normally causes all output on the display to be echoed on LPT1:, will not work if you redirect standard output.

SAMPLE SESSION

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

The main steps in running a program are:

1. Loading BASIC
2. Typing the program
3. Running the program
4. Saving the program
5. Loading the program into memory

Loading BASIC

For this sample, load BASIC by typing:

```
BASIC 
```

Typing the Program

Type in the sample program below. After typing each line, check it for any mistakes. If there are no mistakes, press . If you make a mistake, use the key to move the cursor to the mistake and retype the rest of the line to correct the mistake.

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

Check your program again. If you find a mistake, enter the line number and type the line again. The newly typed line replaces the old line.

It does not matter if you enter Line 15 after Line 20; BASIC still reads and executes Line 15 before “looking” at Line 20. BASIC always reads program lines in numerical order.

Tell BASIC to execute this program by typing:

```
RUN 
```

Your screen should display:

```
WILLIAM SHAKESPEARE WROTE THE MERCHANT OF VENICE
```

BASIC has powerful special keys that let you correct mistakes without retyping the entire line. These commands are discussed in Chapter 4, "General Information."

Saving the Program on Disk

You can save any BASIC program on disk by assigning it a *pathname*. The pathname tells BASIC on which disk and directory you want to save the file and the name of the file. The pathname must be enclosed in quotation marks. Pathnames must conform to the conventions discussed in Chapter 1, "About BASIC for MS-DOS."

For example, to save the program we just wrote on Drive B, in the directory BOOKS with the filename Author.bas, use the following command:

```
SAVE "B:\BOOKS\author.bas" ENTER
```

Notice that BOOKS is located in the root directory since it is preceded by the root symbol (\).

You can also save the file with this command:

```
SAVE "author.wil"
```

which saves the program as Author.wil in the current directory on the MS-DOS current drive.

It takes a few seconds for the computer to find a place on the disk to store a program and to copy the program from memory to the disk. When the program is saved on the disk, BASIC displays its prompt (OK).

Loading the Program Into Memory

If, after writing or running other programs, you want to use this program again, you must load it back into memory from disk.

For example, to load the program Author.bas from the directory BOOKS, type:

```
LOAD "B:\BOOKS\author.bas",R ENTER
```

The R option tells BASIC to run the program after loading it.

```
LOAD "author.wil"
```

loads the file Author.wil from the current directory on the current drive.

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

```
RUN "pathname"
```

RUN automatically loads and runs the program specified by *pathname*.

The SAVE, LOAD, and RUN commands are discussed in more detail in Chapter 10.

GENERAL INFORMATION

When BASIC displays the `OK` prompt, you can type in program lines or commands. If you want BASIC to read what you type in, you must press `ENTER` at the end of the line.

A single line can be a maximum of 255 *visible* characters. Visible characters are those that take up a space on the display.

Since 255 characters cannot fit on one line of the display, BASIC moves the extra characters to the next line. This is called *wrap-around*.

BASIC looks at the first character of a line. If it is a digit, BASIC stores the line in memory as a program line. For example, if you type:

```
10 PRINT "THE TIME IS " TIME$ ENTER
```

BASIC takes this as a program line and stores it in memory. It does not execute the line until you type `RUN` and press `ENTER`.

If the first character is not a digit, BASIC tries to execute the line as a command. For example, if you type:

```
MILES=133:GALLON=11:MPG=MILES/GALLON ENTER
```

BASIC immediately executes this line as a command. After it is executed, the statement no longer exists in memory, but the values of the variables `MILES`, `GALLON`, and `MPG` are stored in memory.

This BASIC capability lets you use the computer as a calculator for quick computations that do not require an entire program.

Editing

BASIC lets you correct errors in program and command lines quickly and efficiently without retyping entire lines.

You can use the special keys defined at the end of this chapter to make corrections or changes at any time. To correct a line, simply use the arrow keys to position the cursor on the line you want to alter. After you make changes to the line, press `ENTER` to store the changes.

When modifying program lines, you can edit specific lines by typing:

```
EDIT line number 
```

If the line number you specify does not exist, BASIC returns an Undefined line number error.

You can also specify the current program line by using a period (.) instead of a line number:

```
EDIT . 
```

The current line is the last line entered, the last line altered, or a line in which an error has occurred. Notice that you must type a space before the period; otherwise, BASIC displays a Syntax error message.

BASIC automatically enters EDIT when a syntax error occurs when executing a program. It displays the line that contains the error and waits for you to make corrections.

Sample Editing Session

This sample session shows how you can easily edit lines in BASIC. Even though the sample is a BASIC program, you can use the same procedure for command lines. All special keys used in this session are described at the end of this chapter.

To begin the sample session, type the following line and press :

```
100 PRINT "This is our example line."
```

Now use the to position the cursor on Line 100.

Use the to move across the line to the T in This. Type lowercase and then . Remember, none of the changes you make to a program line are recorded until you press .

Type LIST 100 and press to see that BASIC has stored your change in memory. BASIC displays:

```
100 PRINT "this is our example line."
```

Notice that you can make simple changes by typing over the old material.

Now, position the cursor over Line 100 again. Press **[END]** and then use **[←]** to position the cursor on the second set of quotation marks. Press **[INSERT]** and type:

We inserted the second sentence. **[ENTER]**

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

Now use the EDIT command to edit Line 100. Type:

EDIT . **[ENTER]**

Remember, the period (.) tells BASIC to edit the current line. Don't forget to type a space before the period.

Using **[TAB]** and **[←]**, position the cursor on the i in inserted. Hold down **[CTRL]** and press **[END]**. BASIC deletes all the characters you have inserted except we and the blank space.

Press **[BACKSPACE]** to delete the space.

Hold down **[CTRL]** and press **[←]** to position the cursor on the preceding word. Press **[DELETE]** twice to delete we. Press **[INSERT]** and then **["]** to put the quotation mark at the end of the statement.

Press **[ENTER]** to record the changes. You can use the LIST command to see the new line.

Use the EDIT command again, this time with the line number. Type:

EDIT 100 **[ENTER]**

Using **[←]**, position the cursor on the P in PRINT. Press the space bar to change the P to a blank.

Press **[CTRL]** while pressing **[←]** to position the cursor on the t in this. Press **[T]** to change the lowercase t to a capital T.

Instead of pressing **[ENTER]** after you make the changes, press **[ESC]**. Use the LIST command. Notice that BASIC did not record your changes because you pressed **[ESC]** instead of **[ENTER]**. The **[ESC]** key tells BASIC to erase the line and not to make any changes to the line.

Now you have used most of the special keys in the editor. If you still do not feel comfortable with them, go through the sample session again.

If you feel confident that you understand the editor, read on to learn about some special keys that make it easier and faster to change lines anywhere on the screen.

Special Keys

The following keys perform special functions in BASIC for entering and editing lines. To use some of these keys you must press and hold down the **CTRL** key while pressing the second key. For example, when you use **CTRL****H** to backspace, hold down the **CTRL** key and press **H** at the same time.

Key	Description
CAPS	switches to all uppercase or uppercase/lowercase mode.
SPACEBAR	changes the current character to a blank and advances the cursor 1 position to the right.
BACKSPACE or CTRL H	backspaces the cursor, erasing the first character to the left. All characters to the right move left 1 position. Use this to correct typing errors before you press ENTER .
CTRL BREAK or CTRL C	interrupts line entry and starts over with a new line. Any changes previously made to the line are not saved.
ENTER or CTRL M	ends the current line. BASIC reads the line.
ESC	erases the entire line from the screen, but not from memory.
CTRL L or CTRL HOME	clears the screen and positions the cursor at the first position in Row 1.
DELETE	deletes the character at the cursor position and moves all remaining characters to the left 1 position.
INSERT or CTRL R	turns the insert mode on if it is off, or off if it is on. The insert mode lets you add new characters to the line at the cursor position.

<code>HOME</code> or <code>CTRL K</code>	moves the cursor to the first position in Row 1.
<code>END</code> or <code>CTRL N</code>	moves the cursor to the end of the line.
<code>CTRL END</code> or <code>CTRL E</code>	deletes all characters from the current cursor position to the end of the line.
<code>TAB</code> or <code>CTRL I</code>	advances the cursor to the next tab position. Tab positions are set at every 8 characters.
<code>←</code> or <code>CTRL J</code>	moves the cursor 1 position to the left.
<code>→</code>	moves the cursor 1 position to the right.
<code>↑</code> or <code>CTRL G</code>	moves the cursor up 1 row to the character above the current cursor position.
<code>↓</code> or <code>CTRL ↓</code>	moves the cursor down 1 row to the character below the current cursor position.
<code>CTRL ←</code> or <code>CTRL B</code>	moves the cursor to its left and to the first character in the preceding word, which is the first character preceded by a blank.
<code>CTRL →</code> or <code>CTRL F</code>	moves the cursor to its right and to the first character in the next word, which is the first character preceded by a blank.
<code>CTRL G</code>	rings the bell at the terminal.
<code>CTRL J</code>	issues a linefeed. This moves the cursor to the next line of the display without executing or storing the line.

The following special keys act differently while BASIC is executing programs or commands:

<code>CTRL NUM LOCK</code>	pauses execution. Press any key to continue.
<code>CTRL BREAK</code>	terminates execution and returns you to BASIC's prompt.
<code>ENTER</code> or <code>CTRL M</code>	signifies the end of data entry. When a BASIC program or command prompts you to enter data, press <code>ENTER</code> to end the response.

The **ALT** Key

The **ALT** key provides a quick and easy way to type certain BASIC keywords. These keywords are associated with alphabetic characters (A-Z).

To enter these keywords, press and hold down the **ALT** key while pressing the desired letter. BASIC inserts the keyword at the current cursor position. The keywords and their associated letters are listed below.

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

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

The **PRT SC** Key

Pressing **SHIFT PRT SC** dumps the current text content of the screen to the line printer LPT1:.

CTRL PRT SC is the line printer echo key, which acts as a toggle switch. If the echo is off, pressing **CTRL PRT SC** once causes all characters sent to the screen to also be sent to the line printer LPT1:. Pressing **CTRL PRT SC** a second time turns off this echo feature.

BASIC CONCEPTS

This chapter describes the different ways BASIC handles and manipulates data. By understanding how BASIC does this, you can build more efficient programs.

Elements of a Program

A *program* is a group of instructions that performs a certain task. It is made up of 1 or more numbered *lines*.

Each line can contain a maximum of 255 visible characters. Of the 255 characters, BASIC automatically reserves 1 space for each digit in the line number and another space for the space following the line number. If you enter more than 255 visible characters, BASIC truncates the line.

Here is a sample program line:

```
10 PRINT "one"
```

A *line number* is always the first element of a program line. In BASIC line numbers must be in the range 0 to 65529. In the sample program line, the line number is 10.

A BASIC *statement* follows the line number. A statement tells BASIC to perform a specific operation. In the sample program line, the statement is `PRINT "one"`. This statement tells BASIC to print, or display, the word `one` on the screen.

You can have more than 1 statement on a program line by placing a colon between each statement. For example:

```
20 FOR X = 1 TO 5:PRINT "one":NEXT X
```

This program line has 3 statements. They are:

1. `FOR X = 1 TO 5`
2. `PRINT "one"`
3. `NEXT X`

You can add explanations, or *remarks*, to your program lines. A remark is preceded by a single quotation mark to separate it from the statements. Here is a program line with a remark:

```
20 FOR X = 1 TO 5:PRINT "one":NEXT X 'loop
```

Data

Data is information on which BASIC performs its operations. Data can be numbers, characters, or symbols. BASIC classifies data into two groups: string and numeric.

String data is a sequence of ASCII characters, graphics or non-ASCII symbols. A string can be a maximum of 255 characters. If the string is entered on a program or command line, it must be enclosed in quotation marks. (See "Constants" later in this section.) If the string is entered in response to a prompt, it is not enclosed in quotation marks. BASIC does not evaluate string data; it simply stores it for the program to use or manipulate.

Hint: ASCII stands for American Standard Code for Information Interchange. In ASCII, each character has a unique number that represents it. This is necessary since computers understand and process only numbers.

Here are some sample strings:

"JIM"	"MAIN STREET"	"255 CENTRAL AVE"
"25 dollars"	"\$250"	"2 + 4"

Notice that numbers can be in a string. Remember, BASIC does not evaluate strings. Type the following line at BASIC's prompt:

```
PRINT "2 + 4"
```

BASIC does not add 2 and 4. It obeys the command PRINT and displays 2 + 4 on your screen.

Strings use 3 bytes of memory plus the number of characters in the string. For example, the string "CATS" takes up 7 bytes of memory: 4 for the string plus 3.

Numeric data consists of positive and negative numbers. BASIC divides numeric data into 5 groups: integer, single precision, double precision, hexadecimal, and octal.

Integers are whole numbers in the range -32768 to +32767 that do not contain a decimal point. For example:

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

Integers use the least amount of memory (2 bytes). Therefore, BASIC can access them fastest.

Single precision numbers can be a maximum of 7 digits and may have a decimal point. Single precision numbers must be in the range 10^{-38} to 10^{+38} . Sample single precision numbers are:

10.001 -200034 123.4567

If a single precision number is more than 7 digits, BASIC displays the number in scientific notation, or exponential format, in the E form. For example:

1.74E 6.98E8 104E-7

BASIC stores a single precision number in 4 bytes of memory.

Double precision numbers can include a maximum of 16 digits and may have a decimal point. Double precision numbers have the same range as single precision numbers. Sample double precision numbers are:

1010234567 -8.7777651010

If a double precision number is more than 16 digits, BASIC displays the number in scientific notation, or exponential format, in the D form. For example:

8.00100708D12 -6.7765499824D16

BASIC stores double precision numbers in 8 bytes of memory. Although double precision numbers consume more memory, they are the most exact.

Hexadecimal numbers are the hexadecimal representation of decimal numbers. They contain 1 to 4 digits and are preceded by &H. The hexadecimal numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Here are some hexadecimal numbers and their decimal equivalents:

Hex	Decimal
&H76	118
&H02FF	767
&HFF	255

BASIC stores hexadecimal numbers as integers.

Octal numbers are the octal representation of decimal numbers. They contain 1 to 6 digits and are preceded by &O or &. Although only the & is required, we recommend that you use &O for clarity in your programs. The octal numbers are 0, 1, 2, 3, 4, 5, 6, and 7. Here are some octal numbers and their decimal equivalents:

Octal	Decimal
&7	7
&O123	83
&O000456	302

BASIC stores octal numbers as integers.

Constants

Constants are values input to a program that are not subject to change. Constants can be either string or numeric data (integer, single or double precision, hexadecimal, or octal).

Numeric data that will not change can be represented as either a string or numeric constant. If you use punctuation in the number, it must be a string constant. For example:

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

When BASIC encounters a data constant in a statement, BASIC must determine how to store it:

If the value is enclosed in quotation marks, BASIC stores it as a string.

If the value is not enclosed in quotation marks, BASIC stores it as an integer or a single precision or a double precision number, according to the requirements described in the previous section. The section, "Declaring Numeric Constants and Variables," describes ways to override BASIC's classification of constants.

BASIC evaluates numeric constants in program lines as soon as you enter the line. It does not wait until you run the program. If any numbers are out of range for their type, BASIC returns an error message immediately.

Here are some examples of constants:

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

This line contains 4 string constants: NAME, ADDRESS, CITY, and STATE. These values will not change. Every time BASIC executes this statement, the same 4 words are printed.

```
PRINT "1000 PLUS"; 2000; "EQUALS";3000
```

The 1000 is a string constant, the 2000 and the 3000 are numeric constants.

Variables

Variables are symbolic names for a value in a BASIC program. A variable name can be a maximum of 40 characters and must begin with a letter (A-Z).

Note: You cannot use any of the reserved words listed in Appendix A as variable names. However, reserved words may be imbedded in a variable name.

The following are examples of variable names:

A	A1	ADDRESS	ADDRESS.OLD
L	L2	LEN2	LENGTH

The 2 types of variables are string and numeric. BASIC initially classifies all variables as single precision with a value of zero (0). (The next section describes how to declare variables as string, integer, or double precision variables.)

The following examples assign a value to a variable.

```
LET A = 12345
A = 601.432
BALANCE = 338.92
```

BASIC automatically stores all the above examples as single precision numbers. Chapter 10, "BASIC Keywords," describes more ways to assign values to variables.

Declaring Numeric Constants and Variables

BASIC lets you override its automatic classification of numeric constants and variables.

Numeric Constants

To change the way BASIC stores a numeric constant, add one of the following symbols to the end of the number. If BASIC must shorten a number to meet the new requirements, it rounds the number.

- ! declares a single precision number. For example, BASIC stores the number 12.345678901234! as a single precision number: 12.34568.
- E declares the number a single precision exponential number. For example, BASIC stores the number 1.2E5 as a single precision number: 120000.
- # declares a double precision number. For example, BASIC stores the number 1.5# as a double precision number: 1.5. BASIC does not expand constants when declaring them double precision.
- D declares the number a double precision exponential number. For example, BASIC stores the number 1.2D2 as a double precision number: 120.

See the next section on converting numbers for important information on converting from numbers to another precision.

Numeric Variables

BASIC initially classifies all numeric variables as single precision. You can declare variables as other than single precision in 2 ways:

- Append a symbol to the variable name:
 - % declares an integer variable. BASIC stores the value of the variable as an integer. I%, FT%, and COUNTER% are samples of integer-declared variables.
 - ! declares a single precision variable. BASIC stores the value of the variable as a single precision number. F!, NM!, and BALANCE! are samples of variables declared as single precision.

- # declares a double precision variable. BASIC stores the value of the variable as a double precision number. S#, AD#, and YTDTOTAL# are samples of variables declared as double precision.
- \$ declares a string variable. The value of the variable must be enclosed in double quotes. A\$, WRD\$, and CITY\$ are samples of variables declared as string variables.

Note: Any variable name can represent 4 different variables. For example, A5%, A5!, A5#, and A5\$ are all valid and distinct variable names.

- Use the following BASIC statements:

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

Chapter 10 describes these BASIC statements fully.

Numeric Precision Conversion

Your program may ask BASIC to convert numeric data from one precision to another. The following section describes this procedure.

When converting single/double precision to integers, BASIC rounds the fractional portion of the number, if any. For example:

A% = 32766.7	BASIC stores as 32767
A% = -123.4567	BASIC stores as -123

When converting integers to single/double precision, BASIC appends a decimal point and zeroes to the right of the original value. For example:

A# = 32767	BASIC stores as 32767.000000000000
A! = -1234	BASIC stores as -1234.000

When converting double to single precision, BASIC rounds the number to 7 significant digits. For example:

A! = 1.2345678901234567 BASIC stores as 1.234568
A! = 1.3333333333333333 BASIC stores as 1.333333

When converting single to double precision, BASIC adds trailing zeroes to the right of the original value. If the original value has an exact binary representation in single precision format, the resulting value is accurate. For example:

A# = 1.5 BASIC stores as 1.5000000000000000

However, if a number does not have an exact binary representation, the conversion creates an erroneous value. For example:

A# = 1.3 BASIC stores as 1.299999952316284

You should not use such conversions in your program because most fractional numbers do not have exact binary representations. You can avoid this by forcing the constant to be double precision, such as:

A# = 1.3# or A# = 1.3D

which BASIC stores as 1.3.

If you must convert from single to double precision, the following programs show a special technique.

Type and run the following program:

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

BASIC prints 1.299999952316284

Now type and run this program:

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

BASIC prints 1.3. Converting the single precision number to a string before converting it to a double precision value causes BASIC to store the value accurately.

Note: BASIC cannot automatically convert numeric data to string data or vice versa. This results in a `Type mismatch` error. Use the `VAL` and `STR$` functions to accomplish this kind of conversion.

Manipulating Data

BASIC uses *expressions* as a way to manipulate data. An expression is 2 or more pieces of data connected by operators.

An *operator* is a symbol or a word that signifies some action to be performed on the specified data. Each data item is called an *operand*.

An expression might look like this:

operand1	operator	operand2
6	+	2

A few operators allow only one operand, for example:

operator	operand
-	5

Expressions must be used in a BASIC statement, such as:

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

BASIC has four types of operators:

Arithmetic	used for numeric data only.
String	used for string data only.
Relational	used for both numeric and string data.
Logical	used for numeric data only.

Arithmetic Operators

Arithmetic operators perform operations on numeric data. Both operands must be numeric. When BASIC evaluates the expression, all operands are converted to the same degree of precision, that of the most precise operand. The result of the arithmetic operation is also returned to this degree of precision.

The arithmetic operators are listed below. They are in order of precedence, that is, the order in which BASIC executes them if 1 or more operators are in the same statement.

- ^ Exponentiation. Calculates the power of a number. For example, 2^3 is 8 (2 to the power of 3 is the same as $2*2*2$).
- Negation or Unary Minus. Makes a number negative. For example, -10 is "negative ten."
- *, / Multiplication, Division. For example, $3*3$ is 9, and $10/5$ is 2.
- \ Integer Division. BASIC rounds both operands to integers and truncates the result to an integer. Integer division is faster than standard division. For example, $10\backslash 4$ is 2.
- MOD Modulus Arithmetic. BASIC performs integer division as described above and returns the remainder as an integer value. For example, $10 \text{ MOD } 3$ results in 1.
- +, - Addition, Subtraction. For example, $2+9$ is 11, and $15-8$ is 7.

String Operator

The *string operator* is the plus sign (+). It appends one string to another. All operands must be strings, and the resulting value is 1 string. For example:

```
PRINT "APRIL SHOWERS " + "BRING" + " MAY  
FLOWERS."
```

prints APRIL SHOWERS BRING MAY FLOWERS.

Relational Operators

Relational operators compare 2 pieces of numeric data or 2 pieces of string data. The result of the comparison is either *true* or *false*. If the relationship is true, BASIC returns -1. If the relationship is false, BASIC returns 0 (zero).

The relational operators are, in order of precedence:

- = Equal. Both operands are equal.
- < Less Than. The first operand is less than or precedes the second operand.
- > Greater Than. The first operand is greater than or follows the second operand.
- >< or <> Inequality. The operands are not equal.
- <= or =< Less Than or Equal To. The first operand is less than (precedes) or is equal to the second operand.
- >= or => Greater Than or Equal To. The first operand is greater than (follows) or is equal to the second operand.

Relational operators are usually used within an IF/THEN statement. For example:

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

BASIC looks at the value in variable A. If the value is equal to 1, BASIC prints the word `CORRECT`.

If arithmetic and relational operators are combined in the same expression, BASIC evaluates the arithmetic operations first. For example:

```
IF X*Y/2 <= 15 PRINT "AVERAGE SCORE"
```

BASIC performs the arithmetic operation `X*Y/2` and then compares the result with 15.

When relational operators are used with strings, BASIC compares the strings character by character. When it finds 2 characters that do not match, it checks to see which character has the lower value ASCII code. The character with the lower ASCII code comes before the word with the higher ASCII value in an alphabetical listing, just as one word comes before another in a dictionary.

Consider these examples:

```
"A" < "B"
```

BASIC compares the ASCII value of the 2 strings. The ASCII value for A is 65, and the ASCII value for B is 66. Since 65 is less than 66, BASIC returns a -1. BASIC displays the result if you type PRINT and the expression, for example, PRINT "A">"B".

```
"CODE" > "COOL"
```

This is false. The first 2 characters of the strings match. However, the third character does not. BASIC then compares the ASCII codes. The ASCII code for D is 68 and the code for O is 79. Since 79 is not less than 68, BASIC returns a 0.

```
"TRAIL" < "TRAILER"
```

This is true. If BASIC reaches the end of one string before finding 2 characters that don't match, the shorter string is considered the less of the two strings (lower in precedence). Therefore, TRAIL is the lesser of the two strings.

Also note that leading blanks are significant in string comparisons. Therefore, " A" comes before "A" because the ASCII code for blank is 32 and the ASCII code for A is 65.

Logical Operators

Logical operators, or *Boolean operators*, make logical comparisons of numeric values. The logical operators are NOT, AND, OR, XOR, EQV, and IMP. They take a set of true/false values, usually from relational expressions, and return a true or false result.

The following table describes the result for each logical operator given the described true/false values.

Operator	Meaning of Operation	First Operand	Second Operand	Result
NOT	The result is the opposite of the operand.	1		0
		0		1
AND	When both values are true, the result is true. Otherwise, the result is false.	1	1	1
		1	0	0
		0	1	0
		0	0	0
OR	When both values are false, the result is false. Otherwise, the result is true.	1	1	1
		1	0	1
		0	1	1
		0	0	0
XOR	When one of the values is true, the result is true. Otherwise, the result is false.	1	1	0
		1	0	1
		0	1	1
		0	0	0
EQV	When both values are true or both values are false, the result is true.	1	1	1
		1	0	0
		0	1	0
		0	0	1
IMP	The result is true unless the first value is true and the second value is false.	1	1	1
		1	0	0
		0	1	1
		0	0	1

Normally, logical operators are used in IF/THEN statements. For example:

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

BASIC prints the variable X if 1 relational expression is true or if both are true. If both are false, BASIC does not print the variable X.

```
IF S$ = "TEXAS" AND C$ = "AUSTIN" THEN PRINT Z$
```

BASIC prints the value of Z\$ if S\$ contains the word TEXAS and C\$ contains the word AUSTIN.

You may also use logical operators to make bit comparisons of 2 numeric expressions. In this case, BASIC does a bit-by-bit comparison of the 2 values, according to predefined rules for the specific operator. Note that the operands are converted to integer type, stored internally as 16-bit, two's complement numbers. This information is important when doing bit comparisons.

Hierarchy of Operators

BASIC uses a predefined hierarchy when performing operations on expressions with multiple operators. This list shows the operators in the order that BASIC would perform the operations in a statement. Remember, BASIC evaluates statements from left to right. Operators with the same level of hierarchy are shown on the same line.

```
^
unary -
* /
\
MOD
+ -
< > = <= >= <>
NOT
AND
OR XOR
EQV
IMP
```

Consider this expression:

$X * X + 5^{2.8}$

BASIC evaluates 5 to the 2.8 power first, then multiplies $X * X$, and finally adds the 2 values.

You can change the order of the hierarchy by adding parentheses to an expression. BASIC always evaluates the expressions inside the parentheses before evaluating the rest of the expression. Look at this expression:

$X * (X + 5)^{2.8}$

BASIC evaluates the expression $(X + 5)$ first and raises that value to the 2.8 power before performing the multiplication.

If an expression contains multiple parentheses, BASIC evaluates the innermost parentheses first.

Functions

A *function* is a built-in sequence of operations that BASIC performs on data. BASIC always performs functions first when evaluating a statement.

Numeric functions, such as ABS, SQR, and COS, perform predefined operations on numeric data.

String functions, such as MID\$, VAL\$, and LEN\$, perform operations on string data.

Functions are described in Chapter 10.

ARRAYS

An *array* is a group of related data values stored consecutively in memory. The entire group of data values is referred to by one variable name. Each data value is called an *element* of the array. A *subscript* is an integer used to refer to each element of the array. For example, an array named A may contain 3 elements referred to as:

A(1) A(2) A(3)

Note: Normally array elements start with Element 0; however, to simplify this example, it starts with Element 1. You can modify the initial array element base-number with the `OPTION BASE` statement.

You can use each of these elements to store a separate data value, such as:

A(1) = .10
A(2) = .20
A(3) = .30

You can imagine an array as a row of boxes, with the numbers on them to identify them. Each box can hold a different value. For example, Array A may hold your expenses.

A(1)	A(2)	A(3)
Grocery Expense	Gas Expense	Clothes Expense

Array A

This is a 1-dimensional array, because elements are arranged in a single row and only one subscript is used to an element. For example, A(1) holds your grocery expense.

This program creates a 1-dimensional array:

```
5  CLS:OPTION BASE 1
10 DATA GROCERY,GAS,CLOTHES
20 DIM A(3)
30 FOR C = 1 TO 3
40 READ NAMES$
50 PRINT "ENTER THE "NAMES$" EXPENSE IN DOLLARS"
60 INPUT A(C)
70 NEXT C
```

The DIM statement in Line 10 reserves space in memory for an array named *A* with 3 elements. As you enter the expenses, the grocery expense is stored in *A*(1), the gas expense in *A*(2) and the clothes expense in *A*(3).

Add these lines to the program to print the contents of Array *A*:

```
100 RESTORE
110 FOR C = 1 TO 3
120 READ NAMES$
130 PRINT:PRINT NAMES$ " = " A(C)
140 NEXT C
```

Use RUN to see the results of this program.

You can add more dimensions to the array such as storing the expenses by weeks.

	Col 1 Grocery	Col 2 Gas	Col 3 Clothes
Row 1 Week 1			
Row 2 Week 2			
Row 3 Week 3		$A(3,2) =$ Gas expense for Week 3	
Row 4 Week 4			

This is a 2-dimensional array. Each element is referred to by 2 subscripts:

A(row,column)

For example, A(3,2) points to the third week's gas expense.

To make a 2-dimensional array from the earlier program, add the following lines:

```
25  FOR R = 1 TO 4:RESTORE
75  NEXT R
105 FOR R = 1 TO 4:RESTORE
150 NEXT R
```

and change these lines:

```
20  DIM A(4,3): W = 1
50  PRINT "ENTER THE ";NAMES$;" EXPENSES IN DOL-
    LARS FOR WEEK NO: "W
60  INPUT A(R,C)
70  NEXT C: W = W + 1
100 RESTORE: W = 1
130 PRINT:PRINT NAMES$;" EXPENSE FOR WEEK
    NO: ";W;" = ";A(R,C)
140 NEXT C:W = W + 1
```

Run this program and see how it works. We simply added another subscript to the original array. Now instead of referring to an element by a row number only, we refer to it by both a row and column number.

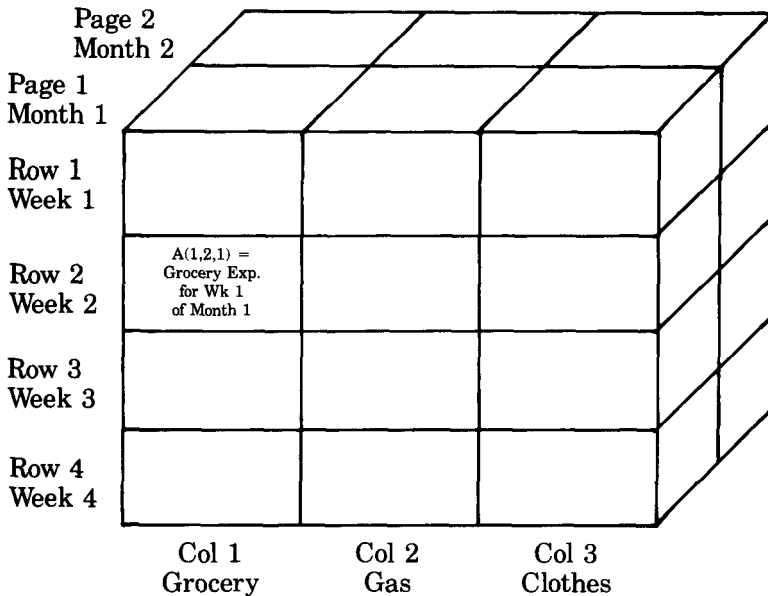
You can add yet another dimension, or subscript, to the array by adding these lines:

```
22  FOR P = 1 TO 2: RESTORE
78  W = 1: NEXT P
102 FOR P = 1 TO 2: RESTORE
160 W = 1:NEXT P
```

and changing these lines:

```
20  DIM A(2,4,3):W = 1: M = 1
50  PRINT "ENTER THE ";NAMES$;" EXPENSE IN DOL-
    LARS FOR WEEK NO: ";W;" OF MONTH NO: ";M
60  INPUT A(P,R,C)
75  NEXT R: M = M + 1
100 RESTORE: W = 1: M = 1
130 PRINT:PRINT NAMES$;" EXPENSE FOR WEEK
    NO: ";W;" OF MONTH NO: ";M;" = ";A(P,R,C)
150 NEXT R: M = M + 1
```

Run the program to see how it works.



Imagine the third dimension as an entirely new page. Here, you refer to an element in the array by using 3 subscripts:

$A(\text{page}, \text{row}, \text{column})$

For example, in $A(1,2,1)$, the first subscript (1) stands for the month. The second subscript (2) stands for the week and the third subscript (1) stands for the Grocery category. So $A(1,2,1)$ contains the Grocery expense for the second week of the first month.

Types of Arrays

Arrays may be of any type: string, integer, single precision, or double precision. You can have a maximum of 255 dimensions in your array and a maximum of 32,767 elements in each dimension.

The amount of memory that an array occupies is equal to the number of bytes it takes to store that type of variable times the number of elements. For example, if you have a double precision array of 30 elements, it occupies 240 bytes of memory. Remember, double precision numbers are stored in 8 bytes of memory.

Defining Arrays

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

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

automatically creates an array named A containing 6 elements and assigns element A(5) the value 300. Use this method only if your array contains fewer than 11 elements (0-10). If your array contains more than 11 elements, you must use the DIM statement.

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

```
DIM C$(99)
```

creates Array C and reserves memory space for 100 double precision elements.

See the DIM statement in Chapter 10 for more information on creating arrays.

DISK FILES

You may want to store data on disk for future use. To do this, you need to store the data in a *file*. A file is an organized collection of related data. It may contain a mailing list, a personnel record, or almost any kind of information.

You access this information in *records*. A record is a small portion of data from the disk file such as a name and address in a mailing list file. A record is the largest block of information that you can address with a single command.

With BASIC you can create and access 2 types of files: sequential access or direct access.

Sequential Access Files

With *sequential access* files, you can access data only in the same order as it was originally stored. To read from or write to a particular section in the file, you must first read through all the records in the file from the beginning until you get to the desired record.

Data is stored in a sequential access file as ASCII characters. Therefore, it is ideal for storing free-form data without wasting space between data items. However, it is limited in flexibility and speed.

The statements and functions used with sequential files are:

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

These statements and functions are discussed in more detail in Chapter 10.

Creating a Sequential Access File

1. To create the file, open it in Output mode (with the letter O) and assign it a *buffer* number in the range 1 to 15. Use either form of the OPEN statement:

```
OPEN "O", 1, "list.dat"  
OPEN "list.dat" FOR OUTPUT AS 1
```

Either of the preceding examples opens a sequential output file named List.dat and gives Buffer 1 access to this file.

2. To input data from the keyboard into 1 or more program variables, use either INPUT or LINE INPUT. For example:

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

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

3. To write data to the file, use the WRITE# statement. (You also can use PRINT#, but be sure you delimit the data.) For example:

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

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

4. To ensure that all the data has been written to the file, use the CLOSE statement. For example:

```
CLOSE 1
```

closes access to the file that uses Buffer 1 (the same buffer used to open the file).

Sample Program

```
10 OPEN "O", 1, "list.dat"
20 LINE INPUT "ENTER A NAME OR 'DONE' TO END >
   ";N$
30 IF N$ = "DONE" THEN 60
40 WRITE# 1, N$
50 PRINT: GOTO 20
60 CLOSE 1
```

The file List.dat stores the data you input through the aid of the program, not the program itself. To save the program above, you must assign it a name. Use the SAVE command as described in Chapter 3. For example, enter SAVE "pay-roll.bas".

Every time you modify a program, you must save it again (you can use the same name); otherwise, the original program remains on disk, without your latest corrections.

5. To access data in the file, reopen it, this time in the Input mode with the letter I. For example:

```
OPEN "list.dat" FOR INPUT AS 1
```

opens the file named List.dat for sequential input, using Buffer 1.

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

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

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

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

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

Sample Program

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

Updating a Sequential Access File

1. To add data to the file, open it in Append mode with the letter A. For example:

```
OPEN "A", 1, "list.dat"
```

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

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

The following program illustrates this technique. It builds upon the file previously created.

Note: Read through the entire program first. If you encounter BASIC keywords that are unfamiliar to you, refer to Chapter 10 for their definitions.

Sample Program

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

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

```
70 OPEN "list.dat" FOR INPUT AS 1
80 IF EOF(1) THEN 200
90 INPUT# 1, N$
100 PRINT N$
110 GOTO 80
200 CLOSE 1
```

After you have run this program, save it. For example, enter SAVE "payroll2.bas" to save the program under a different name than the previous program.

Direct Access Files

With a *direct access* file, you can access data anywhere within the file. It is not necessary to read through all the information, as with a sequential access file, because in a direct access file you can access each record of information individually by its number.

More program steps are required to create and access direct access files, but they are more flexible and easier to update than sequential access files.

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

The maximum number of logical records is 16,777,215. Each record may contain a minimum of 1 and a maximum of 32768 bytes.

The statements and functions used with direct access files are:

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

These statements and functions are discussed in more detail in Chapter 10.

Creating a Direct Access File

1. To create the file, open it for *random access* in Random mode ("R"). For example:

```
OPEN "R", 1, "listing.dat", 32
```

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

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

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

allocates the first 20 positions in Buffer 1 to string variable N\$, the next 4 positions to A\$, and the next 8 positions to P\$. The variables N\$, A\$, and P\$ are now *field names*.

3. To move data into the buffer, use the LSET statement. Numeric values must be converted to 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. For example:

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

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

```
PUT 1, CODE%
```

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

The following program writes information to a direct access file:

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

The 2-digit code that you enter in Line 30 becomes a record number. That record number stores the name(s), amount(s), and phone number(s) you enter when Lines 50, 60, and 70 are executed. The record is written to the file when BASIC executes the PUT statement in Line 110.

After typing this program, save it and run it. Then, enter the following data:

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

BASIC stores SMITH, 34.55, and 567-9000 in Record 20 of file Listing.dat.

Accessing a Direct Access File

1. Open the file in Random mode:

```
OPEN "R", 1,"listing.dat",32
```

2. Use the FIELD statement to allocate space in the buffer for the variables that will be read from the file. For example:

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

3. Before you use the GET statement to read the record, you can check to see if the record is in your file. Set a variable in your program equal to the record size you used in the OPEN statement. LOF returns the length of the file in bytes. The total number of bytes in the file divided by the record size is equal to the largest record number in the file. An attempt to access a record number greater than the largest record number in the file results in an Input past end error.

For example:

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

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

```
GET 1, CODE%
```

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

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

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

The program may now access the data in the buffer.

The following program accesses the direct access file Listing.dat (created with the previous program). When BASIC executes Line 30, enter any *valid* record number from Listing.dat. This program prints the contents of that record.

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

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

```
2-DIGIT CODE, 0 TO END? 20
SMITH
$34.55
567-9000
```

To update Listing.dat, simply use LOAD to load the previous program (the one that created Listing.dat) and run it.

DISPLAYING TEXT AND GRAPHICS

Interpreter BASIC includes many commands that let you display text and graphics images in black and white and in color.

Before using these commands, you must first use the `SCREEN` statement to select 1 of 3 *screen modes*, numbered 0-2. In making your selection, consider all the attributes affected by the screen mode:

- The ability of the screen to display both graphics and text or only text.
- The available colors.
- The size of the screen in points (*resolution*) for the graphics modes.
- The number of characters per line (*text width*).
- The number of pages of video memory for the screen display.

This chapter discusses the above screen mode attributes in detail, and then provides a summary of each mode's attributes. The last section, "Specifying Coordinates," describes how to tell BASIC where on the screen to display graphics images. It is appropriate only if you are in a mode that supports graphics.

Graphics Capability

Screen Mode 0 is the *text-only mode*. When you use it, you **cannot** display graphics.

Screen Modes 1 and 2 are the *graphics modes*. With them, you can display both text and graphics at the same time.

Color

Although Screen Mode 0 limits you to text display, it lets you use a wide range of colors. You can make each character any of 16 colors.

Screen Mode 1 is the 4-color graphics mode. It has 2 sets (*palettes*) of 4 colors each. At any given time, you can select either palette for your text and graphics.

Both Modes 0 and 1 start with a black background and a white foreground. To activate color, provide the appropriate value for the *burst* parameter when you select the screen mode. (See the **SCREEN** statement in Chapter 10.) After activating color, you can change colors by specifying the desired colors with the appropriate numbers. The color numbers and the commands you need vary with the screen modes. They are described in the following sections.

Screen Mode 2 is the black-and-white graphics mode. The background is black, and the foreground is white. Although you cannot **swap** the foreground and background colors, you can achieve much the same result by drawing a new background in white, and then drawing **on** that background in black. (See “Colors in Mode 2,” below, for more information on how to do this.)

Colors in Mode 0

In Screen Mode 0, the following 16 colors are available at all times for your text characters:

Number	Color
0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	white
8	gray
9	light blue
10	light green
11	light cyan
12	light red
13	light magenta
14	yellow
15	high-intensity white

As stated earlier, the foreground is initially white and the background is black. The *border*, the group of points forming the perimeter of the screen, is also initially black. So, it is invisible. To see the border, change either its color or the color of the background.

Once you activate color, you can change colors whenever you want—even from one character to another. To change colors, use the `COLOR/Text` statement as described in Chapter 10. The colors available for various parts of the screen are:

Foreground	Any of the 16 colors, in either non-blinking mode (0-15) or blinking mode (16-31). To figure the number for a blinking color, add 16 to the number given in the preceding table.
Background	Any of Colors 0-7.
Border	Any of Colors 0-15.

Colors in Mode 1

In Screen Mode 1, either of 2 palettes is available at any time for your graphics displays:

Number	Color in Palette 0	Color in Palette 1
0	<i>current background color</i> (starts as black)	<i>current background color</i> (starts as black)
1	green	cyan
2	red	magenta
3	brown	high-intensity white

Once you activate color, you can select a palette and change palettes as often as you wish. To do so, use the `COLOR/Graphics` statement as described in Chapter 10. Use the `COLOR/Graphics` statement to change the background color, as well. (The background can be any of the 16 colors listed earlier for Screen Mode 0.)

To select a **particular color** within the current palette, you specify the color's number when you enter the statement that actually draws the image. See the `DRAW`, `CIRCLE`, and `LINE` statements in Chapter 10 as examples.

Colors in Mode 2

In Screen Mode 2, the available colors are:

Number	Color
0	black
1	white

The background is black, and the foreground is white. If you want to simulate a white background, you can either “white-wash” the screen (using the `PAINT` statement) or draw a box and fill it in (using the `LINE` statement). Simply specify Color 1 when you enter the `PAINT` or `LINE` statement.

Note: Remember that you are **simulating** a white background. The true background color for Screen Mode 2 is always black, and the true foreground color is always white. Keep this in mind when you use statements such as `PSET`, in which BASIC uses the foreground color if you do not specify the color.

See Chapter 10 for information on `LINE`, `PAINT`, `DRAW`, `CIRCLE`, and other statements that you can use to create graphics images in Screen Mode 2.

Resolution

The number of points on a **graphics mode** screen is the resolution. The greater the number of points, the sharper the image. The 2 resolutions are:

Medium resolution	320 x 200	available in Screen Mode 1
High resolution	640 x 200	available in Screen Mode 2

The horizontal length (the length of the x axis) is given first, followed by the vertical length (y axis).

Notice that there are fewer vertical points than horizontal points. The vertical points are farther apart, and it takes fewer of them to make an inch.

Therefore, when you want to draw a square, for instance, you need more points on the horizontal sides of the square. How many more? That depends on the *aspect ratio*, the comparison of points per inch vertically to horizontally.

With a Tandy CM-1 Color Monitor, the aspect ratio for Screen Mode 1 is 5/7. For Screen Mode 2, it is 5/14. If you use a different monitor, the aspect ratio might be different. Use this formula to calculate it:

$$\text{aspect ratio} = \frac{\begin{array}{c} \text{number of} \\ \text{vertical points} \\ \text{in the viewing area} \end{array}}{\begin{array}{c} \text{height of} \\ \text{the viewing area} \end{array}} \div \frac{\begin{array}{c} \text{number of} \\ \text{horizontal points} \\ \text{in the viewing area} \end{array}}{\begin{array}{c} \text{height of} \\ \text{the viewing area} \end{array}}$$

Note: The viewing area is the portion of the screen on which you can draw images. Initially, it is the same size as the screen. If you wish, you can make it smaller using the VIEW statement. When measuring the viewing area, use any unit (inches, centimeters, millimeters, and so on), but be sure to use the same unit for both the height and width.

When drawing graphics, also keep in mind that the number of points per inch varies between the 2 graphics screen modes. Therefore, an image drawn in Screen Mode 1 looks different if drawn with the same *coordinates* in Screen Mode 2 (and vice versa). (For more information on coordinates, see “Specifying Coordinates.”)

Text Width

In Screen Mode 0, you can have either 40 or 80 characters per line. In 40-character width, the characters appear double-sized. To change the width, use the WIDTH statement.

Screen Mode 1 uses 40 characters per line. Screen Mode 2 uses 80. Do not try to change the width to 80 while in Screen Mode 1 or to 40 while in Screen Mode 2. If you do, WIDTH forces the screen into the other screen mode, changing the colors and the resolution.

Video Memory

Regardless of the screen mode, BASIC sets aside 16K bytes of memory for video display use. In Screen Mode 0, it splits this amount into 4 *pages* if the screen width is 80 or into 8 pages if the width is 40. You can store information on one page while displaying another. To do this, specify the *active* and *display* pages when you enter the SCREEN statement. (See Chapter 10.)

Summary

For ease of reference, here is a summary of each screen mode and its attributes:

Screen Mode 0

Graphics Capability:	No
Resolution:	Not applicable
Color Set:	16 colors
Text Width:	40 or 80
Video Page Size:	2048 bytes if width = 40 4096 bytes if width = 80
Max. No. of Pages:	8 if width = 40 4 if width = 80

Screen Mode 1

Graphics Capability:	Yes
Resolution:	320 x 200
Aspect Ratio:	5/7
Color Set:	4 (2 palettes)
Text Width:	40
Video Page Size:	16384 bytes
Max. No. of Pages:	1

Screen Mode 2

Graphics Capability:	Yes
Resolution:	640 x 200
Aspect Ratio:	5/14
Color Set:	black and white
Text Width:	80
Video Page Size:	16384 bytes
Max. No. of Pages:	1

Specifying Coordinates

To draw an image on the screen, you must tell BASIC where on the screen to put the image. For example, when using the LINE statement to draw a line, you need to specify the starting and ending points of the line. You do this by specifying the horizontal and vertical *coordinates* of the points.

The horizontal coordinate is also called the *x-coordinate*, or *x*. The vertical coordinate is the *y-coordinate*, or *y*. Always list *x* before *y* when referring to a point.

Unless you use the WINDOW statement to change them, the coordinates for the graphics modes are as follows:

Screen Mode 1

(0,0)	(319,0)
(0,1)	(319,1)
(0,2)	(319,2)
	
	
	
(0,197)	(319,197)
(0,198)	(319,198)
(0,199)	(319,199)

Screen Mode 2

(0,0)	(639,0)
(0,1)	(639,1)
(0,2)	(639,2)
	
	
	
(0,197)	(639,197)
(0,198)	(639,198)
(0,199)	(639,199)

The coordinates shown above are *absolute* coordinates. Some commands allow you to refer to a point relative to the current point on the screen (the last point referenced). In that case, you specify the number of points from the current point. The coordinates you specify are *relative* coordinates.

As an example, if you use the `CIRCLE` statement to draw a circle, the last point referenced is the center of the circle. If you then execute a `LINE` statement, using relative coordinates of $(0,0)$ as the starting point, BASIC starts the line at the center of the circle.

You can use positive or negative values as relative coordinates. If you specify a negative value, BASIC subtracts it from the coordinate of the last point referenced. If you specify a positive value, BASIC adds it to the coordinate of the last point referenced.

INTRODUCTION TO BASIC KEYWORDS

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

Chapter 10 describes all of BASIC's keywords. This chapter explains the format used in Chapter 10. It also gives a quick summary of all of BASIC's keywords.

Format for Chapter 10

Keyword	Statement Function
Syntax	
Brief definition of keyword.	
Detailed definition of keyword and any parameters or arguments for that keyword.	
Example(s)	
Sample Program(s)	

This format varies slightly, depending on the complexity of each keyword. For instance, some keywords require certain parameters or arguments and others do not.

Some keywords are followed by *defining words* that explain how to use the command. The defining words are:

Communications	used with RS-232 communications
Graphics	must be in a graphics screen mode (Mode 1 or 2)
Trap	used for event trapping

There are more, but they should be self-explanatory.

Some keywords have sample programs that further explain their use or illustrate useful applications that may not be readily apparent.

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

For example, if you type:

DELETE. ENTER

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

Terms Used in Chapter 10

line	A numeric expression that identifies a BASIC program line. Each line has a number in the range 0 to 65529.
integer	Any integer expression. It may consist of an integer or of several integers joined by operators. Integers are whole numbers and may be in the range -32768 to 32767 unless otherwise specified.
string	Any string expression. It may consist of a string, several strings joined by operators, or a string variable. A string is a sequence of characters that is to be taken verbatim.
number	Any numeric expression. It may consist of a number, several numbers joined by operators, or a numeric variable.
dummy number or dummy string	A number (or string) used in an expression to meet syntactic requirements, but the value of which is insignificant.

Statements

A *statement* tells the computer to perform some operation. The following is a brief description of all BASIC statements:

Statement	Description
AUTO	automatically generates line numbers.
BEEP	produces a sound from the computer speaker.
BLOAD	loads a memory image file from disk.
BSAVE	saves a memory image file to disk.
CALL	calls an assembly-language subroutine.
CALLS	calls an MS-FORTRAN subroutine.
CHAIN	loads another program and passes variables to that program.
CHDIR	changes the current directory.
CIRCLE/Graphics	draws an ellipse on the screen.
CLEAR	frees memory for data without erasing the program in memory.
CLOSE	closes access to a disk file.
CLS	clears the screen.
COLOR/Graphics	selects background and palette for Screen Mode 1.
COLOR/Text	selects foreground, background, and border display colors for Screen Mode 0.
COM/Trap	enables communications event trapping.
COMMON	passes variables to a chained program.
CONT	continues program execution.
DATA	stores data in your program so that you can access it with a READ statement.
DEFDBL	defines variables as double precision.
DEF FN	defines a function according to your specifications.
DEFINT	defines variables as integers.
DEF SEG	defines the current segment address.
DEFSNG	defines variables as single precision.
DEFSTR	defines variables as strings.
DEF USR	defines the offset of the entry point for USR routines.
DELETE	removes program lines from memory.
DIM	defines the dimensions of an array.
DRAW/Graphics	draws images on the screen.

Statement	Description
EDIT	edits program lines.
END	ends a program.
ENVIRON	modifies BASIC's Environment String Table.
ERASE	erases an array.
ERL	returns the number of the line in which an error occurred.
ERR	returns an error code after an error.
ERROR	simulates the specified error.
FIELD	organizes a direct access buffer.
FILES	displays names of files and directories on a disk.
FOR/NEXT	establishes a program loop.
GET	gets a record from a direct access file or transfers a specific number of bytes from a communications file.
GET/Graphics	transfers graphics images from the screen to memory.
GOSUB	transfers program control to a subroutine.
GOTO	transfers program control to the specified line.
IF/THEN/ELSE	evaluates an expression and performs an operation if conditions are met.
INPUT	accepts data from the keyboard.
INPUT#	accepts data from a sequential access device or file.
INPUT\$	accepts data from the keyboard or a sequential access file.
IOCTL	sends control data to a device driver.
KEY	assigns or displays the current function-key soft values.
KEY/Trap	enables key-event trapping.
KILL	deletes a disk file.
LCOPY	copies all text data on the screen to the printer.
LET	assigns a value to a variable. (The keyword LET may be omitted.)
LINE/Graphics	draws a line on the display.
LINE INPUT	accepts an entire line from the keyboard.
LINE INPUT#	accepts an entire line from a sequential access file.

Statement	Description
LIST	lists a program to the display or printer.
LLIST	prints a program on the printer.
LOAD	loads a program from disk.
LOCATE	positions the cursor on the screen.
LOCK	prohibits access by other processes to all or part of an opened file.
LPRINT	prints data at the printer.
LPRINT USING	prints data at the printer in a specified format.
LSET	moves data (and left-justifies it) to a field in a direct access file buffer.
MERGE	merges a disk program with a resident program.
MID\$	replaces a portion of a string.
MKDIR	creates a directory.
NAME	renames a disk file.
NEW	erases a program from RAM.
ON COM GOSUB	branches to a subroutine when activity occurs on the communications channel.
ON ERROR GOTO	sets up an error-trapping routine.
ON/GOSUB	evaluates an expression and branches to a subroutine.
ON/GOTO	evaluates an expression and branches to another program line.
ON KEY() GOSUB	branches to a subroutine when a specific key is pressed.
ON PEN GOSUB	branches to a subroutine when the light pen is activated.
ON PLAY() GOSUB	branches to a subroutine when music buffer contains fewer than the specified number of notes.
ON STRIG() GOSUB	branches to a subroutine when a joystick button is pressed.
ON TIMER() GOSUB	branches to a subroutine when timer equals the specified number.
OPEN	opens a disk file.
OPEN "COM	opens a communications file.
OPTION BASE	declares the minimum value for array subscripts.
OUT	sends a byte to a machine output port.

Statement	Description
PAINT/Graphics	fills in an area of the screen with a selected color.
PEN/Trap	controls light pen event trapping
PLAY	plays musical notes.
PLAY/Trap	controls background music event trapping.
POKE	writes a byte into a memory location.
PRESET/Graphics	draws a point in color at a specified position on the screen.
PRINT	lists data to the display.
PRINT USING	lists data to the display in a specific format.
PRINT#	writes data to a sequential access file.
PRINT# USING	writes data to a sequential access file using the specified format.
PSET/Graphics	draws a point on the screen at a specified position.
PUT/Communications	puts a record into a direct access file or transfers a number of bytes to a communications file.
PUT/Graphics	transfers graphics images from the memory to the screen.
RANDOMIZE	reseeds the random number generator.
READ	reads data stored in the DATA statement and assigns it to a variable.
REM	inserts a remark line in a program.
RENUM	renumbers a program.
RESET	closes all open files on all drives.
RESTORE	restores the DATA pointer.
RESUME	resumes program execution after an error-handling routine.
RETURN	returns from a subroutine to the calling program.
RMDIR	removes a directory.
RSET	moves data (and right-justifies it) to a field in a direct access file buffer.
RUN	executes a program.
SAVE	saves a program on disk.
SCREEN	sets the screen attributes (text, medium- or high-resolution) to be used by subsequent statements.

Statement	Description
SHELL	loads and executes another program as a child process.
SOUND	generates a specific tone for a specified length of time.
STOP	stops program execution.
STRIG	enables the STRIG function.
STRIG/Trap	controls joystick event trapping.
SWAP	exchanges the values of variables.
SYSTEM	returns to MS-DOS.
TIMER/Trap	controls timer event trapping.
TROFF	turns off the tracer.
TRON	turns on the tracer.
UNLOCK	allows access by other processes to all or part of an opened file.
VIEW/Graphics	redefines the screen parameters.
VIEW PRINT	creates a text viewport to redefine screen parameters.
WAIT	suspends program execution while monitoring the status of a machine input port.
WHILE...WEND	executes statements in a loop as long as a given condition is true.
WIDTH	sets the number of characters per line for the screen or printer.
WINDOW	changes the physical coordinates of the screen.
WRITE	prints data on the display.
WRITE#	writes data to a sequential file.

Functions

A *function* is a built-in subroutine. You may only use it as part of a statement. Most BASIC functions return numeric or string data.

Function	Description
ABS	returns the absolute value of a number
ASC	returns the ASCII code of a character.
ATN	returns the arctangent of a number.
CDBL	converts a number to double precision.
CHR\$	returns the character of an ASCII code.
CINT	converts a number to an integer.
COS	returns the cosine of a number.
CSNG	converts a number to single precision.
CSRLIN	returns the current row position of the cursor.
CVD	restores data from a direct access disk file to double precision.
CVI	restores data from a direct access disk file to integer.
CVS	restores data from a direct access disk file to single precision.
DATE\$	sets the date or returns the current date.
ENVIRON\$	returns a string from BASIC's Environment String Table.
EOF	checks for end-of-file or an empty communications input queue.
ERDEV	returns the value of a device error.
ERDEV\$	returns the name of a device for device error.
EXP	returns the natural exponent of a number.
FIX	truncates to a whole number.
FRE	returns the number of bytes in memory not being used.
HEX\$	converts a decimal value to a hexadecimal string.
INKEY\$	returns the keyboard character.
INP	returns the byte read from a port.
INSTR	searches for a specified string.
INT	returns the integer value of a number.

Function	Description
IOCTL\$	returns control data from a device driver.
LEFT\$	returns the left portion of a string.
LEN	returns the length of the string.
LOC	returns the current disk file record number or the number of characters in a communications input queue.
LOF	returns the total number of bytes in a disk file or the amount of free space in a communications file input queue.
LOG	returns the natural logarithm of a number.
LPOS	returns the position of the print head in the printer buffer.
MID\$	returns the midportion of a string.
MKD\$	converts a double precision value to a string for writing it to a direct access file.
MKI\$	converts an integer value to a string for writing it to a direct access disk file.
MKS\$	converts a single precision number to a string for writing it to a direct access file.
OCT\$	converts a decimal value to an octal string.
PEEK	returns a byte from a memory location.
PEN	returns the coordinates of the light pen.
PLAY	returns the number of notes in the music buffer.
PMAP	returns the physical or world coordinates.
POINT	returns either the color of a point or current coordinates.
POS	returns the cursor column position on the display.
RIGHT\$	returns the right portion of a string.
RND	returns a random number.
SCREEN	returns the ASCII code for the character stored at a specific position on the screen.
SGN	determines the sign of a number.
SIN	returns the sine of a number.
SPACE\$	returns a string of spaces.

Function	Description
SPC	prints spaces to the display.
SQR	returns the square root of a number.
STICK	returns the coordinates of the joysticks.
STR\$	converts a number to a string.
STRIG	returns the status of the joystick buttons.
STRING\$	returns a string of characters.
TAB	positions the video cursor or the print head at a specified position.
TAN	returns the tangent of a number.
TIME\$	sets the time or returns the current time.
TIMER	returns the number of seconds since midnight.
USR	calls an assembly language subroutine.
VAL	returns the numeric value of a string.
VARPTR	returns an offset for a variable or buffer.
VARPTR\$	returns character form of memory address of a variable.

BASIC KEYWORDS

ABS

Function

ABS(*number*)

Computes the absolute value of *number*.

A number's absolute value is its magnitude without regard to its sign. Absolute values are always positive or zero.

Example

```
PRINT ABS(-66)
```

prints 66, the absolute value of -66.

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

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

Sample Program

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

ASC

Function

ASC(*string*)

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

ASC returns the value as a decimal number. If *string* is null, an illegal function call error occurs.

Example

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

prints 65, the ASCII code for A.

Sample Program

You can use ASC to be sure a program is receiving proper input. Suppose you want to write a program that requires the user to input hexadecimal digits (0-9, A-F). To be sure that only those characters are input, and all other characters are excluded, you can insert the following routine.

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


ATN

Function

ATN(*number*)

Computes the arctangent of *number* in radians.

ATN returns the angle whose tangent is *number*. *Number* must be given in radians.

Unless you specified the /D switch when starting up BASIC, BASIC returns the result as a single precision number.

To convert this value to degrees, use $\text{ATN}(\text{number} \cdot 180/\text{PI})$, where PI equals 3.141593.

Example

```
PRINT ATN(7)
```

prints the arctangent of 7, which is 1.428899.

```
X = ATN(Y/3) * 57.29578
```

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

AUTO

Statement

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

Automatically generates a line number each time you press **ENTER** when typing a program.

Line is the line number with which you want BASIC to start numbering. To start numbering with the current line number, specify a period (.) as *line*. If you omit *line*, BASIC starts with Line 10.

Increment is the increment for BASIC to use to generate the subsequent line numbers. You must precede *increment* with a comma (,). If you want BASIC to use the increment of the last AUTO statement, type the comma but omit *increment*. If you omit *increment* and the comma, BASIC uses 10. If you omit *line* but include *increment*, BASIC begins numbering with Line 0.

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

To turn off AUTO, press **CTRL** **BREAK** or **CTRL** **C**. The current line is canceled, and BASIC returns to command level.

Examples

AUTO

generates line numbers beginning with Line 10 using increments of 10. For example, 10, 20, 30

AUTO 100,50

generates line numbers beginning with Line 100 using increments of 50. For example, 100, 150, 200

AUTO 700,

generates line numbers beginning with Line 700 using the increment of the last AUTO statement, in this case 50. For example, 700, 750, 800

BEEP

Statement

BEEP

Produces a sound at 800 Hz for $\frac{1}{4}$ second from the computer's speaker.

Using the BEEP statement is the same as typing PRINT CHR\$(7).

Example

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

warns the operator with a beep if the variable X is out of range, that is, greater than 20.

BLOAD

Statement

BLOAD *pathname*[,*offset*]

Loads a memory image file into memory. See BSAVE.

A memory image file is a byte-for-byte copy of what was originally in memory. See BSAVE for information about saving memory image files.

Pathname is a standard file specification as defined in Chapter 1.

Offset is an integer in the range 0 to 65535. *Offset* is the number of bytes into the current segment where BASIC loads the image. If you omit *offset*, BASIC uses the offset specified when the file was saved with BSAVE.

If you specify *offset*, BASIC assumes you want to BLOAD at an address other than the one given when the program was saved and uses the current segment address as set by the last DEF SEG statement. Unless you want to load the file into BASIC's data segment, you must execute the DEF SEG statement before the BLOAD statement.

If you used the /M: switch when you loaded BASIC, specify that address as the offset.

If you specify an offset without using a DEF SEG statement or the /M: switch, BASIC loads the file at that offset from BASIC's data segment, destroying BASIC's workspace.

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

See the section "Interfacing With Assembly-Language Subroutines" in Chapter 11 for more information on loading assembly-language programs.

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

Sample Programs

Program 1

```
10 'SAVE A 50 byte image of memory
20 DEF SEG = &H10
30 FOR I = 256 to 306
40 VLU = PEEK (I)
50 LPRINT "AT ADDRESS ";I; "WE HAVE A VALUE
  OF ";VLU
60 NEXT I
70 BSAVE "prog1",0,50
80 PRINT "Now Run Program 2 to verify that the
  contents saved in the file PROG1 match those
  in the printout produced by this program."
```

Program 2

```
10 'Load a 50 byte file into memory and verify
  it
20 DEF SEG = &H10
30 BLOAD "prog1.bas", 0
40 FOR I = 256 to 306
50 VALUE = PEEK(I)
60 LPRINT "AT ADDRESS ";I; "the loaded value
  is ";VALUE
70 NEXT I
```

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

BSAVE

Statement

BSAVE *pathname,offset,length*

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

Pathname is a standard file specification as defined in Chapter 1.

Offset is an integer in the range 0 to 65535. *Offset* is the number of bytes into the current segment where BASIC starts saving.

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

You must specify *pathname*, *offset*, and *length*. If you omit any of them, BASIC returns an error and terminates the save.

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

When you load BASIC, the data segment (DS) register is set to the address of BASIC's workspace. You must execute a DEF SEG statement before executing BSAVE, unless you used the /M: switch when you loaded BASIC. Without the DEF SEG statement or the /M: switch, BASIC's workspace could be destroyed.

Sample Program

See BLOAD.

CALL

Statement

CALL *variable* [(*parameter list*)]

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

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

Parameter list contains the variables that are passed to the external subroutine. The number, type, and length of the parameters being passed must match with the parameters expected by the assembly-language subroutine.

If you omit *parameter list*, BASIC executes an 8086 CALL instruction. Your assembly-language subroutine should return with a simple RET instruction.

When you execute a CALL statement, BASIC transfers control to the subroutine through the address given in the last DEF SEG statement and the segment offset specified by *variable*. See the section “Interfacing With Assembly-Language Subroutines” in Chapter 11 for more details.

Example

```
100 I=45 : J=100 : K = 55
110 myrout = &H0000
120 DEF SEG = &H1700
130 CALL myrout(I,J,K)
```

The subroutine, Myrout, begins at offset 0 in the segment that begins at 1700. The values of I, J, and K are passed to the routine.

CALLS

Statement

CALLS *variables [(parameter list)]*

Transfers program control to a routine written in MS™-FORTRAN. CALLS works just like the CALL statement, except that CALLS passes arguments as segmented addresses.

CALLS uses the address given in most recently executed DEF SEG statement to locate the routine being called.

CDBL

Function

CDBL(*number*)

Converts *number* to double precision.

This function may be useful if you want to force an operation to be performed in double precision, even though the operands are single precision or integers.

Sample Program

```
210 A=454.67
220 PRINT A, CDBL(A)
```

When run, this program prints the following:

```
454.67      454.67000134277344
```

CHAIN

Statement

CHAIN [MERGE] *pathname* [, [*line*] [, ALL]
[, DELETE *line-line*]]

Lets the current program load and execute another program named *pathname*.

Pathname is a standard file specification as defined in Chapter 1. It specifies the program you want to chain. The program must have been previously saved in ASCII format. See SAVE.

Line is either a line number or a variable containing a line number that specifies where BASIC is to begin execution in the chained program. *Line* is always preceded by a comma (.). If you plan to use the ALL or DELETE options and do not specify a line number, you must specify a comma for *line*. This keeps BASIC from evaluating ALL and DELETE as variables. If you omit *line*, BASIC begins execution at the first program line of the chained program.

The ALL option tells BASIC to pass every variable in the current program to the chained program. If you omit ALL, the current program must contain a COMMON statement to pass variables to the chained program. If chained programs chain subsequent programs and pass variables, each new program must contain either the ALL option or the COMMON statement.

The MERGE option overlays the lines of the chained program with the current program. See the MERGE statement to understand how BASIC overlays (merges) program lines.

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

Examples

```
CHAIN "prog2"
```

loads Prog2, chains it to the program currently in memory, and begins executing it.

```
CHAIN "subprog.bas", , ALL
```

loads, chains and executes Subprog.bas. The values of all the variables in the current program are passed to Subprog.bas.

Sample Program 1

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
COMMON TO PASS VARIABLES.
20 REM SAVE THIS MODULE ON DISK AS "PROG1.BAS"
USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$( ),B$( )
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED "
60 A$(2)="VALUES BEFORE CHAINING"
70 B$(1)="" : B$(2)=""
80 CHAIN "prog2.bas"
90 PRINT : PRINT B$(1) : PRINT : PRINT B$(2) :
PRINT
100 END
```

Save this program as Prog1.bas, using the A option (Enter: SAVE "prog1.bas", A). Enter NEW, and then enter the following program.

```
10 REM THE STATEMENT "DIM A$(2),B$(2)" MAY ONLY
BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS MODULE.
30 REM SAVE THIS MODULE ON THE DISK AS
"PROG2.BAS" USING THE A OPTION.
40 COMMON A$( ),B$( )
50 PRINT: PRINT A$(1);A$(2)
60 B$(1)="NOTE HOW THE OPTION OF SPECIFYING A
STARTING LINE NUMBER"
70 B$(2)="WHEN CHAINING AVOIDS THE DIMENSION
STATEMENT IN 'PROG1'."
80 CHAIN "prog1.bas",90
90 END
```

Save this program as Prog2.bas, using the A option. Load Prog1.bas and run it. Your screen should display:

```
VARIABLES IN COMMON MUST BE ASSIGNED VALUES
BEFORE CHAINING.

NOTE HOW THE OPTION OF SPECIFYING A STARTING
LINE NUMBER

WHEN CHAINING AVOIDS THE DIMENSION STATEMENT IN
'PROG1'.
```

Sample Program 2

Enter NEW and this program:

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
   THE MERGE AND ALL OPTIONS.
20 A$="MAINPROG.BAS"
30 CHAIN MERGE "overlay1", 1000, ALL
40 END
```

Save this program as Mainprog.bas, using the A option. Enter NEW, and then type:

```
1000 PRINT A$;" HAS CHAINED TO OVERLAY1.BAS."
1010 A$ = "overlay1.bas"
1020 B$ = "overlay2.bas"
1030 CHAIN MERGE "overlay2.bas", 1000, ALL,
   DELETE 1020 - 1040
1040 END
```

Save this program as Overlay1.bas, using the A option. Enter NEW, and then these lines:

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

Save this program as Overlay2.bas, using the A option. Load Mainprog.bas and run it. Your screen should display:

```
MAINPROG.BAS HAS CHAINED TO OVERLAY1.BAS.
OVERLAY1.BAS HAS CHAINED TO OVERLAY2.BAS.
```

Hints:

- The CHAIN statement with the MERGE option leaves the files open and preserves the current OPTION BASE setting.
- The CHAIN statement without the MERGE option does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.
- When using the MERGE option, place user-defined functions before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.
- CHAIN automatically executes a RESTORE before running the chained program. The next READ statement starts at the first item of the first DATA statement.

CHDIR

Statement

CHDIR *pathname*

Changes the current directory.

Pathname is a standard directory specification as defined in Chapter 1.

Examples

```
CHDIR "B:\ACCTS\RECVBLE"
```

changes the current directory on Drive B to ACCTS\RECVBLE.

```
CHDIR "\RECORDS"
```

changes the directory on the current drive to RECORDS.

```
CHDIR ".."
```

changes the current directory to the parent directory of the current directory.

CHR\$

Function

CHR\$(code)

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

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

See Appendix B for a list of ASCII codes.

Example

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

prints the character corresponding to ASCII code 35, which is #.

Sample Program

The following program lets you investigate the effect of printing ASCII codes on the display.

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

CINT

Function

CINT(*number*)

Converts *number* to integer representation.

Number must be in the range -32768 to 32767.

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

See also FIX and INT, which also return integer values.

Examples

```
PRINT CINT(1.56)
```

prints 2.

```
PRINT CINT(-1.67)
```

prints -2.

CIRCLE/Graphics

Statement

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

Draws an ellipse on the screen with the specified center and radius.

(*x,y*) specify the coordinates for the center of the circle. *x* is the horizontal coordinate and *y* is the vertical coordinate.

Color indicates the color of the ellipse and must be a valid number in the current color set.

The STEP option tells BASIC that the (*x,y*) coordinates are relative to the last point referenced.

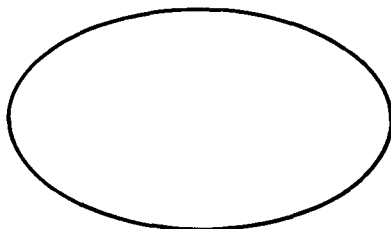
The possible ranges for *x*, *y*, and *color* depend upon the current screen mode as defined in Chapter 8, "Displaying Text and Graphics."

Radius is the major axis of the ellipse.

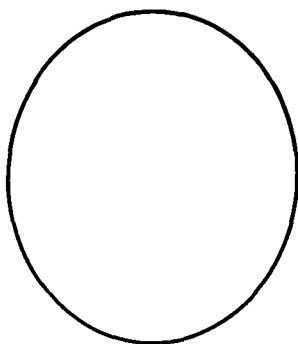
Start and *end* are the beginning and ending angles in radians and must be in the range -6.283186 to 6.283186, or $-2 \cdot \text{PI}$ to $2 \cdot \text{PI}$, where PI equals 3.141593. If you specify a negative *start* or *end* angle, the ellipse is connected to the center point with a line, and the angles are treated as if they were positive.

Aspect is the ratio of the x-radius to the y-radius in terms of coordinates. If *aspect* is less than 1, *radius* is the x-radius and is measured in points in the horizontal direction. If *aspect* is greater than 1, *radius* is the y-radius and is measured in points in the vertical direction. If you omit *aspect*, BASIC uses the defaults for the current screen mode as defined in Chapter 8. When you use the default, BASIC draws a circle.

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



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



See Chapter 8 for more information on aspect ratio and specifying coordinates.

Examples

```
10 SCREEN 1
20 CIRCLE (150,100),50
```

draws a circle with the center at point 150,100 and a radius of 50.

Sample Program

```
10 SCREEN 1
20 FOR I=0 TO 3
30 CLS
40 CIRCLE (150,100),50,I
50 PAINT (150,100),I
60 FOR Q=1 TO 300:NEXT Q
70 NEXT I
80 SCREEN 0
```

CLEAR

Statement

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

Frees memory for data without erasing the program currently in memory. CLEAR erases all arrays, sets numeric variables to zero and string variables to null, and erases any information set using a DEF statement, such as DEF SEG and DEF FN. CLEAR also turns off the SOUND, PEN, and STRIG functions and resets the music background.

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

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

Stack space also must be an integer. This sets aside memory for temporarily storing internal data and addresses during subroutine calls and during FOR/NEXT loops. If you omit *stack space*, BASIC sets aside 768 bytes or one-eighth of the memory available, whichever is smaller. BASIC displays an Out of memory error if stack space for program execution is insufficient.

Note: BASIC allocates string space dynamically. BASIC displays an Out of string space error if no free memory is left for BASIC.

Examples

```
CLEAR
```

clears all variables and closes all files.

```
CLEAR, 45000
```

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

`CLEAR, 61000, 300`

clears all variables and closes all files; then makes 61000 the highest address BASIC may use to run your programs, and allocates 300 bytes for stack space.

CLOSE

Statement

CLOSE [*buffer*,...]

Closes access to a disk file.

Buffer is the number assigned to the file when you opened it. If you omit *buffer*, BASIC closes all open files.

This command terminates access to a file through the specified buffer. If *buffer* has not been assigned by an OPEN statement, then CLOSE *buffer* has no effect.

Do not remove a disk that contains an open file. Close the file first, because the last records might not have been written yet. Closing the file writes the data, if it hasn't already been written.

Note that CLEAR, END, NEW, RESET, and SYSTEM automatically close all files when executed.

See also OPEN and Chapter 7, "Disk Files."

Examples

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

terminates the file assignments to Buffers 1, 2, and 8. You can now assign these buffers to other files with OPEN statements.

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

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

CLS

Statement

CLS

Clears the screen and returns the cursor to the home position. Home is Row 0, Column 0, or in other words, the upper left corner of the screen.

If a viewport is active, CLS clears only the active viewport. To clear the entire screen, you must use VIEW to redefine the entire screen before using CLS.

Changing the screen mode with SCREEN or changing the width with WIDTH automatically clears the screen. You can also clear the screen by typing **CTRL** **L** or **CTRL** **HOME**.

Sample Program

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

COLOR/Graphics**Statement****COLOR** [*background*] [, [*palette*]]

Selects the background color and the palette for Screen Mode 1.

Background specifies the color of the background. It can be any integer in the range 0 to 15. The available colors are:

Number Color

0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	white
8	gray
9	light blue
10	light green
11	light cyan
12	light red
13	light magenta
14	yellow
15	high-intensity white

Notes: In the graphics modes, the background includes the points that would form the border if you were in the text-only mode.

Screen Mode 1 is referred to as the *4-color mode* because of the number of **foreground** colors you can display at a given time. The name has nothing to do with the colors available for the background.

Palette specifies the palette to use for the foreground colors. It can be either 0 for Palette 0, or 1 for Palette 1. Here is a list of each palette's colors:

Number	Color in Palette 0	Color in Palette 1
0	<i>current background color</i> (starts as black)	<i>current background color</i> (starts as black)
1	green	cyan
2	red	magenta
3	brown	high-intensity white

If you omit either *background* or *palette*, BASIC continues to use the current value for that particular parameter. If you omit **both** parameters, BASIC returns an error.

Examples

```
10 COLOR 9,0
```

sets the background to light blue and selects Palette 0.

```
20 COLOR ,1
```

selects Palette 1, without changing the background.

Sample Programs

```
5 SCREEN 1
10 COLOR 12,1
20 LINE (0,0) - (319,199),1
```

Line 10 selects a light red background and Palette 1. Line 20 draws a cyan diagonal line on the display.

```
5 SCREEN 1
10 COLOR 3,0
20 LINE (0,0) - (319,199)
```

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

COLOR/Text**Statement****COLOR** [*foreground*][,*background*][,*border*]

Selects the display colors for the foreground, background, and border for displaying in text mode. To be in text mode, you must have selected Screen Mode 0 with the SCREEN statement.

Foreground is an integer in the range 0 to 31, specifying the foreground color and whether or not it is blinking.

Number

Non-Blinking	Blinking	Color
0	16	black
1	17	blue
2	18	green
3	19	cyan
4	20	red
5	21	magenta
6	22	brown
7	23	white
8	24	gray
9	25	light blue
10	26	light green
11	27	light cyan
12	28	light red
13	29	light magenta
14	30	yellow
15	31	high-intensity white

Background is an integer in the range 0 to 7, specifying the background color. (See the preceding table.)

Note: If you set *foreground* the same as *background*, the characters are invisible.

Border is an integer in the range 0 to 15, specifying the border color. (See the preceding table.)

If you omit any parameter, BASIC continues to use the current value for that particular parameter. If you omit **all** parameters, BASIC returns an error.

Examples

`COLOR 0,7`

selects black characters on a white background.

`COLOR 1,0`

selects blue characters on a black background.

`COLOR 4,0`

selects red characters on a black background.

COM

Statement

COM(*channel*) *action*

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

Channel selects Communications Channel 1 or 2.

Action may be any of the following:

ON	enables communications trapping.
OFF	disables communications trapping.
STOP	temporarily suspends communications trapping.

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

The COM() ON statement turns on the trap. BASIC checks after every program statement to see if a character has come into the communications channel. If so, BASIC transfers program control to the line number specified in the ON COM() GOSUB statement.

The COM() STOP statement temporarily halts communications trapping. If activity occurs on the communications channel, BASIC does not transfer program control to the ON COM() GOSUB statement until you turn on communications trapping again by executing a COM() ON statement. BASIC remembers that activity took place and branches to the subroutine immediately after communications trapping is turned on again.

The COM() OFF statement turns off communications activity trapping. BASIC does not remember if activity took place when communications trapping is turned on again.

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

See ON COM() GOSUB for more information about communications trapping.

Example

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

Line 10 turns on a communications trap on Channel 1. If characters are received on the communications channel, program control transfers to the subroutine beginning at Line 100. If there is no activity on the communications channel, Line 20 prints a message, and Line 40 keeps the program in a loop until there is activity on the communications channel.

COMMON

Statement

COMMON *variable*[,*variable*,...]

Reserves space for *variables* so that they can be passed to a chained program.

Both programs in the chain should contain a COMMON statement. COMMON may appear anywhere in a program, but we recommend using it at the beginning.

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

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

See the CHAIN statement for more information on passing variables.

Example

```
90  DIM D(50)
100 COMMON A, B, C, D(),G$
110 CHAIN "prog3", 10
```

Line 100 reserves space for variables A, B, C, D, and G\$ so that they can be passed to the CHAIN command in Line 110.

CONT

Statement

CONT

Resumes program execution.

You may only use CONT if the program has been stopped by **CTRL BREAK** or the execution of a STOP or an END statement.

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

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

See the STOP statement to terminate execution and the GOTO statement to begin execution at a specific line number.

Example

```
10 INPUT "ENTER 3 NUMBERS a,b,c";A, B, C
20 K=A^2
30 L=B^3/.26
40 STOP
50 M=C+40*K+100: PRINT M
```

Run this program. BASIC prompts for 3 numbers. Type:

```
1, 2, 3 ENTER
```

The computer displays Break in 40. You can now enter a BASIC statement as a command. For example:

```
PRINT L ENTER
```

displays 30.76923. You can also change the value of A, B, or C. For example, to change the value of C, type:

```
C = 4
```

Now type:

```
CONT ENTER
```

and BASIC displays 144.

COS

Function

COS(*number*)

Computes the cosine of *number*.

COS returns the angle (in radians) whose sine is *number*.

Number must be given in radians. If *number* is in degrees, you can convert it to radians by using $\text{COS}(\text{number} \cdot \text{PI}/180)$, where PI equals 3.141593.

BASIC always returns the result as a single precision number unless you specified the /D switch when starting up BASIC.

Examples

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

prints the arithmetic (not trigonometric) difference of the 2 cosines.

```
Y = COS(X * .0174533)
```

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

CSNG

Function

CSNG(*number*)

Converts *number* to single precision.

BASIC rounds the number when converting it to single precision.

Example

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

prints .1453885

Sample Program

```
280 V# = 876.2345678#  
290 PRINT V#, CSNG(V#)
```

When run, this program prints:

```
876.2345678      876.2346
```


CSRLIN

Function

CSRLIN

Returns the current row position of the cursor.

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

Example

```
10 PRINT "This is Line":  
20 PRINT CSRLIN
```

CVD, CVI, CVS

Function

CVD(8- byte string)

CVI(2-byte string)

CVS(4-byte string)

Converts string values to numeric values.

These functions restore data to numeric form after it is read from the disk. Typically, the data has been read by a GET statement and is stored in a direct access file buffer.

CVD converts an 8-byte string to a double precision number.

CVS converts a 4-byte string to a single precision number.

CVI converts a 2-byte string to an integer.

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

Examples

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

assigns the numeric value of GROSSPAY\$ to the double precision variable A#.

Sample Program

This program reads from the file Test.dat, which is created in the sample program for the MKD\$, MKI\$, and MKS\$ functions.

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

BASIC prints 3000, 3000.1 and 3000.00001.

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

DATA

Statement

DATA *constant* [,*constant*,...]

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

This statement may contain as many *constants* (separated by commas) as can fit on a line (a maximum of 255 characters including the word DATA, commas, and spaces).

DATA statements may appear anywhere that is convenient in a program. BASIC reads DATA statements sequentially, starting with the first constant in the first DATA statement and ending with the last item in the last DATA statement.

String constants containing delimiters, such as leading or trailing blanks, colons, or commas, must be enclosed in double quotation marks when used in DATA statements.

The data types in a DATA statement must match with the variable types in the corresponding READ statement, otherwise, BASIC displays a `Syntax error`.

Note that numeric expressions are not allowed in a DATA statement.

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

Examples

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

stores 5 string data items. Quotation marks are not needed since the strings contain no delimiters and the leading blanks are not significant.

```
DATA 2.72, 3.14, 0.01745, 57.29578
```

stores 4 numeric data items.

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

stores both types of constants. Quotation marks are required around the first and third items because they contain commas.

Sample Program

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

This program reads string and numeric data from the DATA statement in Line 30.

DATE\$

Function

DATE\$[= *string*]

Sets the date or retrieves the current date.

String is a literal, enclosed in quotation marks, that sets the current date by assigning a value to DATE\$. If you omit *string*, BASIC retrieves the current date.

Setting the Date

BASIC uses the same dates as MS-DOS, January 1, 1980 to December 31, 2099. You may use either a slash or a hyphen to separate the month, day, and year. You may use any of the following forms to set the current date:

<i>mm/dd/yy</i>	<i>mm/dd/yyyy</i>
<i>mm-dd-yy</i>	<i>mm-dd-yyyy</i>

The month (mm) may be any number 01-12.

The day (dd) may be any number 01-31.

The year (yy or yyyy) may be 01-99 or 1980-2099.

You may omit leading zeroes for the month and day. If you only supply 2 digits for the year, BASIC precedes these digits with 19.

Retrieving the Date

Regardless of the form you use to set the date, BASIC retrieves the date in the following form:

mm-dd-yyyy

The month and day are always returned as 2 digits, BASIC inserts zeroes as necessary.

Examples

```
DATE$ = "9/6/84"  
DATE$ = "9/6/1984"  
DATE$ = "9-6-84"  
DATE$ = "9-6-1984"
```

All the above set the current date as 09-06-1984.

```
PRINT DATE$
```

prints the current system date.

```
CURDATE$ = DATE$
```

assigns the value of the current date to the variable CURDATE\$.

DEFDBL/INT/SNG/STR Statement

DEFDBL *letter*[,*letter*,...]

DEFINT *letter*[,*letter*,...]

DEFSNG *letter*[,*letter*,...]

DEFSTR *letter*[,*letter*,...]

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

You may specify *letter* as a range of letters, such as A-J.

Remember, a type declaration tag always takes precedence over a DEF statement.

Examples

```
DEFDBL L-P
```

classifies all variables beginning with the letters L through P as double precision variables.

```
DEFSTR A
```

classifies all variables beginning with the letter A as string variables.

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

classifies all variables beginning with the letters I through N, W, and Z as integer variables.

```
DEFSNG I, Q-T
```

classifies all variables beginning with the letters I or Q through T as single precision variables.

DEF FN

Statement

DEF FN*name* [(*argument list*)] = *expression*

Defines *name* as a function according to the *expression*.

Name must be a valid variable name. The type of variable you use determines the type of value the function returns. For example, if you use a single precision variable, the function returns single precision values. This name, preceded by FN, is the name of the function when you call it.

Argument list is a list of dummy variables used in *expression*. They are replaced on a one-to-one basis with the variables or values given when the function is called. If you enter several variables, separate them with commas. These variables do not affect variables in your program with the same name.

Expression defines the operation to be performed. A variable used in a function definition may or may not appear in *argument list*. If it does, BASIC uses the value given when the function is called to perform the function. Otherwise, it uses the current value of the variable.

Once you define and name a function (by using this statement), you can use it as you would any BASIC function.

Examples

```
DEF FNR = RND (1)*89+10
```

defines a function FNR to return a random value in the range 10 to 99. Notice that the function can be defined with no arguments.

```
210 DEF FNW# (A#,B#)=(A#-B#)*(A#-B#)
220 I# = 345.998
230 J# = 150.667
240 T = FNW#(I#,J#)
250 PRINT T
```

defines function FNW# in Line 210 using dummy variables A# and B#. Line 240 calls the function and replaces variables A# and B# with variables I# and J#, which are used in the program.

DEF SEG**Statement****DEF SEG**[= *address*]

Assigns the current segment address. The segment address is used by BLOAD, BSAVE, CALL, PEEK, POKE, and USR.

Address is a number in the range 0 to 65535, and may be specified as an integer or a hexadecimal value. If you specify a number outside this range, BASIC returns an illegal function call error and uses the previously set address. If you omit *address*, BASIC sets the current segment address to its data segment (DS).

If you specify *address*, do so on a 16-byte boundary. BASIC shifts the value to the left 4 bits, which is the same as multiplying it by 16 decimal (10 hexadecimal).

Note: BASIC does not check the validity of the resultant segment + offset address.

When you load BASIC, the data segment (DS) register is set to the address of BASIC's workspace. You must, therefore, execute a DEF SEG statement before executing BLOAD, BSAVE, PEEK, POKE, USR, or CALL (unless you used the /M: switch when you loaded BASIC). Without the DEF SEG statement or the /M: switch, BASIC's workspace could be destroyed.

If you execute a DEF SEG to change the DS register, you must execute another DEF SEG to restore the DS register to BASIC's data segment (DS).

Separate DEF and SEG with a space. Otherwise, BASIC interprets it as the variable DEFSEG.

See the section "Interfacing with Assembly-Language Subroutines" in Chapter 11 for more information.

Example

```
10 DEF SEG=&HB800 'Set segment to &B800 Hex
20 DEF SEG        'Restore to BASIC data segment
```

sets the DS register to B8000 hexadecimal (B800H * 10H), which is its default value.

DEF USR

Statement

DEF USR[*number*] = *offset*

Defines the user number and segment offset of a subroutine to be called by the USR function.

Number may be an integer in the range 0 to 9. If you omit *number*, BASIC assumes USR0.

Offset is an integer in the range 0 to 65535. BASIC computes where the subroutine begins in memory by adding the *offset* to the current segment address as set by DEF SEG. BASIC transfers control to this address when you execute the USR function.

If the subroutine is not in BASIC's data segment, you must execute a DEF SEG statement before the USR function.

A program may contain any number of DEF USR statements, allowing access to as many subroutines as necessary. However, only 10 definitions may be in effect at one time.

See the section "Interfacing with Assembly-Language Subroutines" in Chapter 11 and USR in this chapter for more details.

Examples

```
DEF USR3 = &H0020  
DEF SEG = &H1700
```

USR3 begins at 20H bytes into the current data segment which is set at 1700 hexadecimal. When your program calls USR3, control branches to your subroutine beginning at absolute address 17020. ($1700 \times 10 + 20$).

DELETE

Statement

DELETE *line1-line2*

Deletes *line1* through *line2* of the program in memory.

If you omit *line1*, BASIC deletes from the beginning of the program. If omit *line2*, BASIC deletes to the end of the program.

If you specify a line number that does not exist, BASIC displays an `Illegal function call error`.

You can substitute a period (.) for either *line1* or *line2* to indicate the current line number.

Examples

```
DELETE 70
```

deletes Line 70 from memory.

```
DELETE .-110
```

deletes from the current line to Line 110, inclusive.

```
DELETE -40
```

deletes all program lines up to and including Line 40.

```
DELETE 150-
```

deletes program lines starting at and including 150 to the end of the program.

DIM **Statement**

DIM *array*(*dimension*)[,*array*(*dimension*),...]

Sets aside storage for arrays with the dimensions you specify.

Array is the variable name of the array. It may be a string, integer, single precision, or double precision variable.

Dimension is 1 or more integer numbers separated by commas that define the dimensions of the array. The lowest element in a dimension is always zero, unless an OPTION BASE 1 statement is executed.

When you execute the DIM statement, BASIC reserves space in memory for each element of the array. Each element is initially set to zero for numeric arrays or null for string arrays.

If you do not dimension an array, the maximum number of elements it can have is 11 (0-10).

Remember that arrays are completely independent of variables that have the same name; that is MN and MN() are unique.

For more information on arrays, see Chapter 6.

Examples

```
DIM AR(100)
```

sets up a 1-dimensional array AR(), containing 101 elements: AR(0), AR(1), AR(2),..., through AR(100).

```
DIM L1%(8,25)
```

sets up a 2-dimensional array L1%(), containing 9 x 26 integer elements.

DRAW/Graphics**Statement****DRAW** *string*

Draws an image on the screen.

String specifies 1 or more of the movement commands listed below. *String* must be enclosed in quotation marks.

Movement commands

Each of the following movement commands begins movement from the current graphics position, which is the coordinate of the last graphics point plotted with another graphics command, such as LINE or PSET. The current position defaults to the center of the screen if no previous graphics command has been executed.

U [<i>n</i>]	Moves up <i>n</i> points.
D [<i>n</i>]	Moves down <i>n</i> points.
L [<i>n</i>]	Moves left <i>n</i> points.
R [<i>n</i>]	Moves right <i>n</i> points.
E [<i>n</i>]	Moves diagonally up and right <i>n</i> points.
F [<i>n</i>]	Moves diagonally down and right <i>n</i> points.
G [<i>n</i>]	Moves diagonally down and left <i>n</i> points.
H [<i>n</i>]	Moves diagonally up and left <i>n</i> points.
M <i>x,y</i>	Moves to point <i>x,y</i> . If you precede <i>x</i> with a plus (+) or minus (-) sign, DRAW assumes it is a relative position. Otherwise, it is an absolute position.

Prefix Commands

The following prefix commands can precede the movement commands. Prefix commands must be enclosed in quotation marks.

B	plots no points after move.
N	returns to original position when move is complete.
A <i>angle</i>	sets angle of move. <i>Angle</i> may be in the range 0 to 3 (0 = 0 degrees, 1 = 90 degrees, 2 = 180 degrees, and 3 = 270 degrees).
C <i>color</i>	sets color as described in Chapter 8, "Displaying Text and Graphics."

Prefix Commands (Continued)

- Pcolor, border** sets the color to paint and border color at which to stop painting. Possible colors are described in Chapter 8, "Displaying Text and Graphics."
- Sfactor** sets scale factor. *Factor* is an integer in the range 1 to 255. The scale factor is *factor* divided by 4. For example, if *factor* is 2, the scale factor is 2/4. To determine the actual travel distance, multiply the scale factor by the *number* in the movement commands. If you do not specify a factor, BASIC uses 4, which sets the scale to 1.
- TAngle** moves at the specified angle. *Angle* is in the range -360 to +360. If *angle* is positive, movement is counterclockwise. If *angle* is negative, movement is clockwise.
- Xvariable;** executes a substring. The X command lets you execute a second substring from the first string, much like the GOSUB statement. *Variable* is a string variable in your program that contains the substring you want to execute. *Variable* may contain an X command to execute another substring. The semicolon after *variable* is required.

In the prefix commands, the numeric arguments can be constants or variables. If you use a variable name as a numeric argument, you must follow it with a semicolon.

Sample Programs

```
5 SCREEN 1
10 U$ = "U30;"; D$ = "D30;"; L$ = "L40;"; R$ =
   "R40;";
20 BOX$ = U$ + R$ + D$ + L$
30 DRAW "XBOX$;"
40 A$=INKEY$ : IF A$=""THEN 40
50 SCREEN 0
```

draws a rectangle on the screen.

```
5  SCREEN 1
10  U$ = "U30;"; D$ = "D30;"; L$ = "L40;"; R$ =
    "R40;";
20  DRAW "XU$; XR$; XD$; XL$;";
30  A$=INKEY$: IF A$=""THEN 30
40  SCREEN 0
```

draws the same rectangle as the previous example.

```
10  SCREEN 1
20  DRAW "L40 E20 F20"
30  A$=INKEY$: IF A$=""THEN 30
40  SCREEN 0
```

draws a triangle on the screen.

EDIT

Statement

EDIT *line*

Enters the Edit mode. BASIC displays *line* for editing.

You can substitute a period (.) for *line* to indicate the current line number.

See Chapter 4, "General Information," for more information on editing and special keys.

Examples

```
EDIT 100
```

enters the Edit mode at Line 100.

```
EDIT .
```

enters the Edit mode at current line.

END**Statement****END**

Ends program execution and closes all files.

You may place this statement anywhere in the program. It forces execution to end at some point other than the last sequential line.

An END statement at the end of a program is optional.

Sample Program

```
40 INPUT S1, S2
50 GOSUB 100
55 PRINT H
60 END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

Line 60 prevents program control from continuing through the subroutine. Line 100 may be accessed only by a branching statement, such as GOSUB in Line 50.

ENVIRON

Advanced Statement

ENVIRON "*parameter id = text*" [;"*parameter id = text*",...]

Lets you modify BASIC's Environment String Table, such as changing the PATH parameter for a child process or passing parameters to a child process. BASIC's Environment String Table is initially empty.

Parameter id is the name of the parameter.

Text is the new parameter text. It must be separated from *parameter id* by an equal sign (=) or a space. BASIC reads the first nonblank, nonequal sign character after the *parameter id* as the *text*. If you omit *text*, or specify a null string or a semicolon (;), BASIC removes the parameter from the Environment String Table and compresses the table.

Parameter id = text must be enclosed in quotation marks and be typed in all uppercase characters.

When you change a parameter in the Environment String Table, BASIC deletes the old parameter and adds the new one to the end of the table.

If the parameter does not exist in the Environment String Table, BASIC adds it to the end of the table.

For more information on Environment String Tables, see the *Programmer's Reference* manual for your computer (sold separately).

Examples

```
ENVIRON "PATH=A:\"
```

sets the default path to the root directory on Drive A.

```
ENVIRON "SALES=MYSALES"
```

sets the name SALES equal to MYSALES. The Environment String Table now looks like this:

```
PATH=A:\;SALES=MYSALES
```

ENVIRON\$

Advanced Function

ENVIRON\$ [(*"parameter id"*)] [(*number*)]

Returns the specified environment string from BASIC's Environment String Table.

Parameter id specifies the parameter for which to search. ENVIRON\$ returns the text string for *parameter id*. If the parameter does not exist or does not contain a text string, ENVIRON\$ returns an empty string. *Parameter id* must be enclosed in quotation marks. If you omit *parameter id*, you must specify *number*.

Number specifies which parameter to return by its position within the table. ENVIRON\$ returns the text string for the *number* parameter. If there is not a parameter in that position, ENVIRON\$ returns an empty string. If you omit *number*, you must specify *parameter id*.

Parameter id and *number* are mutually exclusive. Only one may be specified on the command line.

For more information on the Environment String Tables, see the *Programmer's Reference* manual for your computer (sold separately).

Example

If you execute the following ENVIRON statements:

```
ENVIRON "PATH=A:\"  
ENVIRON "SALES=MYSALES"
```

the Environment String Table looks like this:

```
PATH=A:\;SALES=MYSALES
```

The command `PRINT ENVIRON$("PATH")` prints `A:\`.

The command `PRINT ENVIRON$(2)` prints `SALES= MYSALES`.

EOF

Function

EOF(*buffer*)

Detects the end of a file.

Buffer is the number assigned to the file when you opened it. It must access an open file.

This function checks to see whether all characters up to the end-of-file marker have been accessed so that you can avoid input past end errors during sequential input.

When used with sequential access files, EOF returns 0 (false), when the end-of-file record has not been read yet, and -1 (true), when it has been read.

When used with direct access files, EOF returns -1 (true) if the last executed GET statement was unable to read an entire record because of an attempt to read beyond the physical end of the file.

Sample Program

The following sequence of lines reads numeric data from Data.txt into the array A(). When the last data character in the file is read, the EOF test in Line 30 is true, so the program branches out of the disk access loop.

```
1470 DIM A(100)    ASSUMING THIS IS A SAFE VALUE
1480 OPEN "I", 1, "data.txt"
1490 I% = 0
1500 IF EOF(1) THEN GOTO 1540
1510 INPUT#1, A(I%)
1520 I% = I% + 1
1530 GOTO 1500
1540 REM PROG. CONT. HERE AFTER DISK INPUT
```

EOF/Communications

Function

EOF(*buffer*)

Detects an empty input queue for communications files.

Buffer is the number assigned to the file when you opened it. It must access an open file.

The value EOF returns depends on the mode (ASCII or binary) in which the file was opened. In ASCII mode, EOF returns a -1 (true) if a CONTROL-Z is received. EOF remains true until the device is closed. In binary mode, EOF returns a -1 (true) when the input queue is empty. EOF becomes false when the input queue is not empty.

Sample Program

These lines are useful in a program when you want to run the program while waiting for communications activity.

```
10 OPEN "COM1:300,N,8,1" AS 1
20 COM(1) ON
30 ON COM(1) GOSUB 1000
.
.
.
1000 "Communication Subroutine Begins Here
.
.
.
1050 IF EOF(1) THEN RETURN
```

Line 10 opens a file for Communications Channel 1 and allocates Buffer 1. Line 30 causes BASIC to perform the subroutine beginning at Line 1000 as soon as there is activity on the communications channel. When all the communications data has been processed, Line 1050 returns to the main program.

ERASE

Statement

ERASE *array[,array,...]*

Erases one or more *arrays* from memory.

This lets you either redimension arrays or use their previously allocated space in memory for other purposes.

If one of the parameters of ERASE is a variable name that is not used in the program, an `Illegal function call` occurs.

Example

```
450 ERASE C,F
460 DIM F(99)
```

Line 450 erases arrays C and F. Line 460 redimensions array F.

ERDEV

Advanced Function

ERDEV

Returns the value of a device error within MS-DOS as set by the Interrupt 24 handler. The lower 8 bits of ERDEV contain the Interrupt 24 error code.

For more information on device drivers and errors, see the *Programmer's Reference* manual for your computer (sold separately).

See also ERDEV\$.

ERDEV\$

Advanced Function

ERDEV\$

Returns the name of the device (as set by the Interrupt 24 handler) when a device error occurs.

If the error occurred on a character device, ERDEV\$ returns the 8-byte character device name.

If the error does not occur on a character device, ERDEV\$ returns the 2-character block device name.

For more information on device drivers and errors, see the *Programmer's Reference* manual for your computer (sold separately).

See also ERDEV.

ERL

Statement

ERL

Returns the number of the line in which an error has occurred.

This function is primarily used inside an error-handling routine. If no error has occurred, ERL returns a 0. If a statement entered at BASIC's prompt causes the error, ERL returns line number 65535 (the largest number that can be represented in 2 bytes).

Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number in variable E.

Sample Program

See ERROR.

ERR	Statement
------------	------------------

ERR

Returns the error code if an error has occurred.

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO.

See Chapter 12 for a list of error codes.

Example

```
IF ERR = 7 THEN 1000 ELSE 2000
```

branches to Line 1000 if the error is an Out of memory error (code 7); if it is any other error, control goes to Line 2000.

Sample Program

See ERROR.

ERROR

Statement

ERROR *code*

Simulates a specified error during program execution.

Code is an integer expression in the range 0 to 255 specifying one of BASIC's error codes.

This statement is used mainly for testing an ON ERROR GOTO routine. When the computer encounters an ERROR statement, it proceeds as if the error corresponding to that code has occurred. (Refer to Chapter 12 for a listing of error codes and their meanings.)

Example

```
ERROR 1
```

causes a NEXT without FOR error (Code 1) when BASIC reaches this line.

Sample Program

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B>5000 THEN ERROR 21 ELSE GOTO 420
400 IF ERR = 21 THEN PRINT "HOUSE LIMIT IS
$5000"
410 IF ERL = 130 THEN RESUME 500
420 S = S+B
430 GOTO 120
500 PRINT "THE TOTAL AMOUNT OF YOUR BET IS";S
510 END
```

This program receives and totals bets until one of them exceeds the house limit.

EXP

Function

EXP(*number*)

Returns the natural exponent of *number*, that is, e (base of natural logarithms) to the power of *number*.

Number must be less than or equal to 88.02968.

This function is the inverse of the LOG function; therefore, *number* = EXP(LOG(*number*)).

BASIC always returns the result as a single precision number unless you specified the /D switch when starting up BASIC.

Example

```
PRINT EXP(-2)
```

prints the exponential value .1353353.

Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE" N "POWER IS" EXP(N)
```

FIELD**Statement**

FIELD *buffer*, *length* AS *variable*[,*length* AS *variable*,...]

Divides a direct access buffer into fields so that you can send data from memory to disk and disk to memory. Each field is identified by *variable* and is the *length* you specify.

Buffer is the number assigned to the file when you opened it.

Variable must be a string variable.

Length is an integer in the range 1 to 255 representing the length of that field. The sum of all field lengths are equal to the record length assigned when you opened the file.

An OPEN statement assigning the buffer number must precede the FIELD statement. FIELD must precede GET and PUT.

You may use the FIELD statement any number of times to re-field a file buffer. Fielding a buffer does not clear the buffer's contents; it only alters the way the buffer is accessed. You may access the same disk file any number of ways simply by re-fielding it.

Note: All data—both strings and numbers—must be placed into the buffer in string form. There are three pairs of functions (MKI\$/CVI, MKS\$/CVS, and MKD\$/CVD) for converting numbers to strings and strings to numbers.

See also Chapter 7, OPEN, CLOSE, PUT, GET, LSET, and RSET.

Examples

```
FIELD 3, 128 AS A$, 128 AS B$
```

BASIC assigns 128-byte fields to the variables A\$ and B\$. If you now print A\$ or B\$, you can see the contents of the field. Of course, this value would be meaningless unless you previously have used GET to read a 256-byte record from disk.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS  
ST$, 7 AS ZP$
```

BASIC assigns the first 16 bytes of Buffer 3 to field NM\$; the next 25 bytes to AD\$; the next 10 to CY\$; the next 2 to ST\$; and the next 7 to ZP\$.

FILES

Statement

FILES [*pathname*]

Displays the names of the files and directories on a disk.

Pathname is a standard file specification as described in Chapter 1.

If you specify *pathname*, BASIC lists all files that match that *pathname*. If you omit *pathname*, BASIC lists all files and directories in the current directory on the current drive. *Pathname* may contain question marks and asterisks as wild cards. See the section on wild cards in Chapter 1 for more information.

If you specify a drive as part of *pathname*, then BASIC lists all files that match the specified *pathname* on **that** drive.

If you omit the filename when specifying *pathname*, BASIC lists all files and directories in the specified directory.

If you omit the path in *pathname*, FILES looks for the file in the current directory.

Examples

```
FILES
```

lists all files and directories in the current directory on the current drive.

```
FILES "\BOOKS\"
```

lists all files in the directory BOOKS.

```
FILES "*.bas"
```

lists all files in the current directory on the current drive with the extension .bas.

```
FILES "pay?????.bas"
```

lists all files beginning with pay followed by any other five or fewer characters, in the current directory on the current drive, with the extension .bas.

FIX

Function

FIX(*number*)

Returns the truncated integer of *number*.

Unlike CINT, FIX does not round the fractional portion of *number* when making it an integer. Instead, FIX simply strips the fractional portion from *number* so that the resultant value is a whole number. The result is the same precision as the argument (except for the fractional portion).

Unlike INT, FIX does not return the next lower number for a negative *number*.

FIX is the same as:

$\text{SGN}(\text{number}) * \text{INT}(\text{ABS}(\text{number}))$.

See also CINT and INT, which also return integer values.

Examples

```
PRINT FIX (2.6)
```

prints 2.

```
PRINT FIX(-2.6)
```

prints -2.

FOR/NEXT**Statement**

FOR *variable* = *initial value* TO *final value* [STEP *increment*]

NEXT [*variable*]

Establishes a program loop that allows a series of program statements to be executed a specified number of times.

Variable must be either integer or single precision. Each FOR/NEXT loop must have a unique variable.

Increment is the number BASIC adds to the *initial value* each time the loop is executed. If you omit *increment*, BASIC increments by 1. If *increment* is a negative value, BASIC decreases the *initial value* each time through the loop. In this case, the *final value* must be less than the *initial value*.

BASIC executes the program lines following the FOR statement until it encounters a NEXT. At this point, it increases *initial value* by the STEP *increment*. If *initial value* is less than or equal to *final value*, BASIC branches back to the line after FOR and repeats the process. When *initial value* is greater than *final value*, the loop is completed, and BASIC continues with the statement after NEXT.

Note: BASIC skips the body of the loop if *initial value* is greater than *final value* when *increment* is positive or if *final value* is greater than *initial value* when *increment* is negative.

Sample Program

BASIC always sets the final value for the loop variable before setting the initial value. For example:

```
820 I=5
830 FOR I = 1 TO I + 5
840 PRINT I;
850 NEXT
```

executes the loop 10 times, which prints:

```
1 2 3 4 5 6 7 8 9 10
```

Nested Loops

FOR/NEXT loops may be nested; that is, a FOR/NEXT loop may be placed within the context of another FOR/NEXT loop.

The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

Sample Program

```
880 FOR I = 1 TO 3
890   PRINT "OUTER LOOP"
900   FOR J = 1 TO 2
910     PRINT "      INNER LOOP"
920   NEXT J
930 NEXT I
```

This program performs 3 outer loops and 2 inner loops within each of the outer loops. BASIC prints the following:

```
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
```

By listing the counter variable, you can use the NEXT statement to close nested loops. (Be sure not to type the variables out of order.) For example, delete Line 920 and change 930 to:

```
930 NEXT J, I
```

If you omit the variables in nested loops, BASIC matches the most recent FOR statement.

FRE

Function

FRE(*dummy argument*)

Returns the number of bytes in memory not being used by BASIC.

Dummy argument can be any string or numeric constant or variable. If you specify a numeric argument, BASIC returns the amount of memory available. If you specify a string argument, BASIC compresses the data before returning the amount of memory available. This frees unused memory that was once used for strings.

BASIC automatically compresses data if it runs out of workspace. This may take a few seconds.

Examples

```
PRINT FRE(44)
```

prints the amount of memory left.

GET

Statement

GET [#]*buffer*[,*record*]

Reads a record from a direct access disk file and places it in the specified *buffer*.

Buffer is the number assigned to the file when you opened it. The number sign is optional. It is provided for compatibility with other BASICs.

Record is an integer in the range 0 to 16,777,215 that specifies which record number you want to access. If you omit *record*, BASIC reads the next sequential record (after the last GET).

When BASIC encounters GET, it reads the record number from the file and places it into the buffer. The actual number of bytes read equals the record length set when the file is opened.

Examples

GET 1

reads the next record into Buffer 1.

GET 1,25

reads Record 25 into Buffer 1.

GET/Communications

Statement

GET [#]*buffer,number*

Transfers data from the communications line to the communications buffer.

Buffer must be the same *buffer* assigned to the file when it was opened. The number sign (#) is optional. It is provided for compatibility with other BASICS.

Number is the number of bytes to transfer. *Number* cannot exceed the value used in the LEN option of the OPEN COM statement.

Note: Because of the low performance associated with telephone line communications, we recommend that you **not** use GET and PUT statements in such applications. Instead, use the other disk I/O statements.

Sample Program

```
10 OPEN "COM1:" AS 1
20 FIELD 1, 8 AS A$
30 OPEN "R",2,"FILE",8
40 FIELD 2, 8 AS B$
50 I = 1
60 GET 1,8
70 PRINT "COMMUNICATIONS BUFFER (A$) =";A$
80 LSET B$=A$
90 PRINT "NOW FILE BUFFER (B$) CONTAINS: ";B$
100 PUT 2,I
110 I = I + 1
120 IF INKEY$ <> "Q" THEN GOTO 60
130 CLOSE
```

This program gets data from Communications Channel 1 and places it in the communications buffer.

GET/Graphics

Statement

GET (x1,y1)-(x2,y2),array

Transfers points from an area on the display to an array.

(x1,y1) specifies the coordinates where the image begins.

(x2,y2) specifies the coordinates where the image ends.

x is the horizontal coordinate and *y* is the vertical coordinate. The ranges for the coordinates depend on the screen mode. See Chapter 8, "Displaying Text and Graphics," for more information.

Array is a numeric array to hold the image. It must be dimensioned large enough to hold the entire image. To ensure that the array is large enough to hold the image, use the following formula:

$$4 + (\text{INT}((h * b + 7)/8) * v)$$

where:

h is the length of the horizontal side of the image.

b the number of bits per point (2 in Screen Mode 1, and 1 in Screen Mode 2).

v is the length of the vertical side of the image.

For example, to store an image that is 10 by 12 in Screen Mode 1, type:

$$4 + (\text{INT}((10 * 2 + 7)/8) * 12) = 40$$

The array must store 40 bytes. The number of bytes per element of an array are 2 for integer, 4 for single precision, and 6 for double precision.

For this example, you need an integer array with 20 elements, a single precision array with 10 elements, or a double precision array with 7 elements.

The information from the display is stored in the array as:

Element 0 the x dimension of the image

Element 1 the y dimension of the image

The remaining elements of the array store the data bits of the image. Numeric data is stored low byte first and then high byte, but the data is transferred high byte first and then low byte.

You use the GET/Graphics and PUT/Graphics statements together for animation and high-speed object motion in the graphics modes. See also PUT/Graphics statement.

Note: GET and PUT work faster in all resolutions if:

$$x \text{ MOD } (8/\text{bits per point}) = 0$$

(See Chapter 5 for an explanation of MOD.)

Sample Program

```
10 DIM A(50,50)
20 SCREEN 1
30 CIRCLE (30,30),20
40 PAINT (30,30)
50 GET (10,0)-(50,50),A
60 PUT (200,100),A
70 END
```

Line 10 sets up an array for storage. Line 20 selects the screen mode. Line 30 draws a circle, and Line 40 fills in the circle. Line 50 gets the circle, and stores it in Array A(). Line 60 retrieves the circle, and puts it on the screen in a new location.

GOSUB

Statement

GOSUB *line*

Branches to the subroutine beginning at the specified line number.

Every subroutine must end with a RETURN. You can call a subroutine as many times as you want. When BASIC encounters a RETURN statement in the subroutine, it returns to the statement that follows the GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement.

The nesting of GOSUB statements is limited only by the amount of memory available in the BASIC stack. If you use CLEAR to increase the amount of memory for the stack, you also increase the maximum number of nested GOSUBs.

Example

```
GOSUB 1000
```

branches to the subroutine at Line 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

transfers control from Line 260 to the subroutine beginning at Line 280. Line 290 instructs the computer to return to the statement immediately following GOSUB.

GOTO

Statement

GOTO *line*

Branches to the specified *line*.

When used alone, GOTO results in an unconditional branch. However, test statements, such as IF/THEN, may precede the GOTO to effect a conditional branch. Note that the GOTO is optional in IF/THEN statements. For example:

```
IF X=0 THEN 360 ELSE 200
```

BASIC branches to Line 360 if X equals 0. If not, BASIC branches to Line 200.

You can use GOTO in the command mode as an alternative to RUN. This lets you pass values assigned as a command to variables used in the program.

Example

```
GOTO 100
```

BASIC transfers control to Line 100.

Sample Program

```
10 READ R
20 IF R = 13 THEN 80
30 PRINT "R=";R
40 A=3.14*R^2
50 PRINT "AREA =";A
60 GOTO 10
70 DATA 5,7,12, 13
80 END
```

Line 10 reads each of the data items in Line 70. Line 60 returns program control to Line 10. This enables BASIC to calculate the area for each of the data items until it reaches item 13.

HEX\$

Function

HEX\$(*number*)

Calculates the hexadecimal value of *number*.

HEX\$ returns a string that represents the hexadecimal value of *number*. Since the value returned is like any other string, you cannot use it in a numeric expression. Thus, you cannot add hex strings. You can concatenate them, though.

Examples

```
PRINT HEX$(30), HEX$(50), HEX$(90)
```

prints the following strings: 1E, 32, and 5A.

```
Y$ = HEX$(X/16)
```

Y\$ is the hexadecimal string representing the integer quotient X/16.

IF/THEN/ELSE

Statement

IF *expression* **THEN** *statement(s)* [**ELSE** *statement(s)*]

Tests a conditional expression and makes a decision regarding program flow.

Expression is any numeric or string expression, usually making logical or relational comparisons.

Statement can be 1 or more valid BASIC statements. If you have more than one statement, separate the statements by colons. You can also specify a line number for BASIC to branch as a statement.

If *expression* is true, BASIC executes the **THEN** *statement*. If *expression* is false, BASIC executes the matching **ELSE** *statement* or the next program line.

You can also use IF/THEN to test the numeric value of a variable. If the variable contains a 0, the expression is true; otherwise, the expression is false.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" : END
```

passes control to PRINT and then to END if X is greater than 127. If X is not greater than 127, BASIC executes the next line in the program, skipping the PRINT and END statements.

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

tests the first expression. If it is true, BASIC prints A < B. Otherwise, BASIC jumps to the ELSE statement and prints B <= A.

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

assigns the value X + 180 to Y if both expressions are true. Otherwise, BASIC executes the next program line, skipping the THEN clause.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN  
400 ELSE 370
```

branches to Line 210 if A\$ is YES. If not, BASIC skips to the first ELSE, which introduces a new test. If A\$ is NO, then BASIC branches to Line 400. If A\$ is any value besides NO or YES, BASIC branches to Line 370.

Sample Program

IF/THEN/ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs. (If the statement does not contain the same number of ELSEs and IFs, each ELSE is matched with the closest unmatched IF.)

```
1040 INPUT "ENTER TWO NUMBERS"; A, B  
1050 IF A <= B THEN IF A < B THEN PRINT A; ELSE  
PRINT " NEITHER"; ELSE PRINT B;  
1060 PRINT " IS SMALLER THAN THE OTHER"
```

This program prints the relationship between the 2 numbers entered.

INKEY\$

Function

INKEY\$

Reads a character in the keyboard buffer, and returns a 0-, 1-, or 2-byte string. INKEY\$ does not echo the character to the display.

- A 0-byte (null) string indicates that no key is pressed.
- A 1-byte string is an actual character read from the keyboard.
- A 2-byte string indicates that the key pressed is one of the special keys that has an extended code. The first byte is hex 00. See Appendices B and D for a complete list of extended codes.

INKEY\$ is invariably put inside some sort of loop. If not, program execution passes through the line containing INKEY\$ before you can press a key.

The **CTRL** **BREAK** and **CTRL** **NUM LOCK** keys are not passed to INKEY\$. Also **ALT** **CTRL** **DELETE**, which does a system reset, is not passed to INKEY\$.

Note: If your program contains an INKEY\$ and you press a function key, BASIC returns 1 character of the key assignment at a time. For example, suppose this statement is executed:

```
A$ = INKEY$
```

Now suppose you press **F1**, which initially has the value LIST. The first time the statement is executed A\$ equals L, the second time A\$ equals I, and so on. Keep this in mind when writing a BASIC routine to trap for a certain key. Your routine may not perform as expected if you accidentally press a function key.

You can assign the result of INKEY\$ to a string variable and test the length of the string to determine whether a 0-, 1-, or 2-character string is returned by INKEY\$. Example:

```
10 A$=INKEY$: IF A$="" THEN 10
20 IF LEN(A$)>1 THEN PRINT ASC(MID(A$,1,1)),
   ASC(MID(A$,2,1)) ELSE PRINT ASC(A$)
30 GOTO 10
```

Example

```
10 A$ = INKEY$  
20 IF A$ = "" THEN 10
```

causes the program to wait for you to press a key.

INP

Function

INP(*port*)

Returns the byte read from *port*.

Port may be any integer from 0 to 65535.

INP is the complementary function of the OUT statement.

Example

```
100 A=INP(255)
```

returns the byte read from port 255 into variable A.

INPUT

Statement

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

Accepts data from the keyboard and inputs it into 1 or more variables. When BASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type something.

Prompt is a string constant that BASIC displays before displaying the question mark prompt. *Prompt* must be enclosed in quotation marks, and follow the keyword INPUT. If, instead of a semicolon, you type a comma after *prompt*, BASIC suppresses the question mark when printing the prompt.

Variable may be 1 or more string or numeric variables to receive the input. If you specify more than 1 variable, separate them by commas.

If INPUT is immediately followed by a semicolon (;), BASIC does not echo the **ENTER** key when you press it as part of a response.

When typing multiple pieces of data on 1 line, separate the data items with a comma. The number of data items you supply must be the same as the number of variables you specify.

Responding to INPUT with too many items or with the wrong type of value (including numeric type) causes BASIC to print the message ?Redo from start. No values are assigned until you provide an acceptable response.

Examples

INPUT Y%

when BASIC reaches this line, you must type any number and press **ENTER** before the program can continue.

INPUT SENTENCE\$

when BASIC reaches this line, you must type in a string. The string does not have to be enclosed in quotation marks unless it contains a comma, a colon, or a leading blank.


```
INPUT "ENTER YOUR NAME, AGE"; N$, A
```

prints the prompt string on the screen, which helps the user enter the right kind of data.

Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT"
  CINT(X * .38) "POUNDS."
```

INPUT#

Statement

INPUT# *buffer, variable[,variable...]*

Accepts data from a sequential device or file and stores it in a program *variable*.

Buffer is the number assigned to the file when you opened it.

Variable is any string or numeric variable to contain the information.

The sequential file may be a disk file, a data stream from a communications device, or the keyboard device.

With INPUT#, data is input sequentially. That is, when the file is opened, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and reopen it.

INPUT# does not care how you place the data in the file—whether you use a single PRINT# statement or 10 different PRINT# statements. INPUT# looks only for the position of the terminating characters and the end-of-file (EOF) marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first nonblank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when BASIC encounters a terminating character or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or a string variable:

Numeric: BASIC ends input when it encounters a carriage return or a comma.

String: BASIC ends input when it encounters a carriage return or a comma, unless the first character is a quotation mark("). If the first character is a quotation mark, BASIC ends input when it encounters a second quotation mark. Thus, a quoted string may not contain a quotation mark as a character.

Examples

INPUT#1, A,B

sequentially inputs 2 numeric data items from the file opened to Buffer 1 and places them in A and B.

INPUT#4, A\$, B\$, C\$

sequentially inputs 3 string data items from the file opened to Buffer 4 and places them in A\$, B\$, and C\$.

INPUT\$

Statement

INPUT\$(*number* [, [#]*buffer*])

Accepts a string of characters from either the keyboard or a sequential access file.

Number is the number of characters to be input. It must be a value in the range 1 to 255.

Buffer is a buffer that accesses a sequential input file. If you include *buffer*, BASIC inputs the string from sequential access file. If you omit *buffer*, BASIC inputs the string from the keyboard. The number sign (#) is optional. It is provided for compatibility with other BASICs.

When inputting the string from the keyboard, BASIC waits until the user enters the number of characters specified by *number*. You do not need to press **ENTER** to signify end-of-line. The character(s) you type are not displayed on the screen. Any character, except **CTRL** **BREAK**, is accepted for input.

When inputting from a sequential file, BASIC inputs the number of bytes specified by *number* from the file assigned to *buffer*.

Examples

```
A$ = INPUT$(5)
```

assigns a string of 5 keyboard characters to A\$. Program execution halts until 5 characters are typed at the keyboard.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A\$. The characters are read from the file associated with Buffer 3.

Sample Programs

This program shows how you can use INPUT\$ to have an operator input a password for accessing a protected file. By using INPUT\$, you can type in the password without anyone seeing it on the video display. To see the full file specification, run the program. When the BASIC prompt returns, enter PRINT F\$.

```
110 LINE INPUT "TYPE IN THE FILENAME.EXT"; F$
120 PRINT "TYPE IN THE PASSWORD -- MUST TYPE 8
CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ + "." + P$
150 PRINT "YOUR FILENAME IS"; F$
```

In the program below, Line 100 opens a sequential input disk file (which we assume has been previously created). Line 200 retrieves a string of 70 characters from the file and stores them in T\$. Line 300 closes the file.

```
100 OPEN "I", 2, "test.dat"
200 T$ = INPUT$(70,2)
300 CLOSE
```

INSTR

Function

INSTR(*[number,]string1,string2*)

Searches for the first occurrence of *string2* in *string1* and returns the position at which the match is found.

Number specifies the position in *string1* to begin searching for *string2*. *Number* must be an integer in the range 1 to 255. If you omit *number*, INSTR starts searching at the first character in *string1*.

If BASIC finds *string2* in *string1*, it returns the starting position of the match; otherwise, it returns zero. If the entire substring is not contained in the search string, BASIC returns a zero.

Examples

Suppose A\$ = "LINCOLN"

Statement	BASIC returns
INSTR(A\$, "INC")	2
INSTR(A\$, "12")	0
INSTR(A\$, "LINCOLNABRAHAM")	0

For a slightly different use of INSTR, try:

```
10 X=INSTR (3, "1232123", "12")
20 PRINT X
```

which prints 5, because the search started at the third character.

Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA statements. It counts the number of addresses with a specified county zip code (761--) and returns that number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
390 READ ADDRESS$
395 IF ADDRESS$ = "$END" THEN 410
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER
   = COUNTER + 1 ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX
ADDRESSES IS" COUNTER: END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX
*76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD
*20760"
440 DATA "1000 TWO TANDY CENTER, FORT WORTH, TX
*76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY,
RICHARDSON, TX *75080"
460 DATA "$END"
```

INT

Function

INT(*number*)

Converts *number* to the largest integer that is less than or equal to *number*.

Number is not limited to the integer range - 32768 to 32767.

The result has the same precision as *number* (except for the fractional portion).

Unlike CINT, INT does not round positive numbers. It does, however, round negative numbers.

Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT (-12.11)
```

prints -13.

IOCTL

Advanced Statement

IOCTL [#]*buffer,string*

Sends a control data string to a device driver. Control data can be sent to a drive only after it has been opened.

Buffer is the number assigned to the driver when you opened it. The number sign (#) is optional. It is provided for compatibility with other BASICs.

String is a string expression containing a series of commands called *control data*. The commands are generally 2 to 3 characters long and may be followed by an alphanumeric argument. The commands are separated by semicolons (;). *String* may be a maximum of 255 bytes.

For more information on device drivers, see the *Programmer's Reference* manual for your computer (sold separately).

Example

If you write your own driver to replace PRN to set the page length, the IOCTL command may be:

PL*n* where *n* is the new page length.

To open the new PRN driver and set the page length at 56 lines per page, use the following statements:

```
10 OPEN "PRN" FOR OUTPUT AS 1
20 IOCTL 1,"PL56"
```

IOCTL\$

Advanced Function

IOCTL\$([#]buffer)

Returns the control data string from a device driver that you have opened previously.

Buffer is the number assigned to the driver when you opened it. The number sign (#) is optional. It is provided for compatibility with other BASICs.

You can use the IOCTL\$ function to confirm that a IOCTL statement succeeded (or failed). You can also use IOCTL\$ to get information from the device.

For more information on device drivers, see the *Programmer's Reference* manual for your computer (sold separately).

Example

```
10 OPEN "\DEV\PRN" AS 1
20 IF IOCTL$(1) = "NR" THEN PRINT "PRINTER NOT
READY"
```

KEY/Set/Display**Statement****KEY** *number,string***KEY ON****KEY OFF****KEY LIST****KEY** *number,string*

Assigns or displays function key values.

Number is an integer in the range 1 to 10 that indicates the function key being defined (**F1** – **F10**).

String is the string expression assigned to the key and may contain a maximum of 15 characters.

You can program the function keys on your computer to generate a specific string of characters. When you press the key, BASIC displays the string on the screen just as if you had typed every character. Initially, the function keys have these values:

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

You can use the **KEY** statement to redefine the function keys so that BASIC displays the strings you use most often.

You can remove the string from a function key by assigning it a string length of zero (""). For example:

```
KEY 1, ""
```

Key **F1** no longer has a string assigned to it. Moreover, assigning a null string (length zero) to a function key disables it as a soft key. If the disabled soft key is pressed, **INKEY\$** returns a 2-byte string. See Appendix D for more information.

KEY ON

KEY ON displays the function key assignment values on Line 25 of the screen. If the screen width is 40, the screen shows 5 of the function key assignments. If the width is 80, the screen shows all 10 assignments. In both cases, the screen shows only the first 6 characters of the string. When you load BASIC, KEY ON is the initial default value.

KEY OFF

KEY OFF erases the soft key assignments from Line 25. The assignments are still active, but the screen does not display them.

BASIC reserves Line 25 for the function key display. Even if the display is turned off, BASIC does not display program lines on Line 25.

KEY LIST

KEY LIST displays all 15 characters of all 10 soft key assignments on the screen.

Note: If your program contains an INKEY\$ and you press a function key, BASIC returns 1 character of the key assignment at a time. For example, suppose this statement is executed:

```
A$ = INKEY$
```

Now suppose you press **F1**, which initially has the value LIST. The first time the statement is executed A\$ equals L, the second time A\$ equals I, and so on. Keep this in mind when writing a BASIC routine to trap for a certain key. Your routine may not perform as expected if you accidentally press a function key.

KEY/Trap**Statement****KEY(number) action**

Turns on, turns off, or temporarily halts key trapping for a specified key.

Action may be any of the following:

ON	enables key trapping
OFF	disables key trapping
STOP	temporarily suspends key trapping


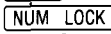




Number may be a number in the range 1 to 20, indicating the number of the key to trap. Function keys use their corresponding function key number (1-10). The cursor direction key trap numbers are:

	11
	12
	13
	14

User-defined keys are 15-20. Use the following syntax to define your own user keys:

KEY number, CHR\$(key) + CHR\$(scan)

Key is one of or a combination of the following:

&H40	 lock key
&H20	 key
&H08	 key
&H04	 key
&H02	Left  key
&H01	Right  key

Scan is the scan code for a physical key on the keyboard. See the appendices for a list of scan codes.

Notes:

- Trapped keys do not go into the keyboard buffer.
- Defining a function key or a cursor-direction key has no effect. BASIC considers them pre-defined.

- You can trap any key, including `CTRL C`, `CTRL BREAK`, and `CTRL ALT DELETE` (the soft boot). This feature makes it possible to prevent BASIC application users from control-breaking out of a program or from accidentally resetting the computer.

You can use the `KEY/Trap` statement in a key trapping routine with the `ON KEY() GOSUB` statement to detect when a specific key is pressed.

The `KEY() ON` statement turns on key trapping for a specific key. BASIC checks after each program statement to see if the specified key has been pressed. If so, BASIC transfers program control to the line number specified in the `ON KEY() GOSUB` statement. For example:

```
KEY(3) ON
ON KEY(3) GOSUB 1000
```

BASIC turns on a trap for `F3`. BASIC continues to execute the other program statements, checking after each statement to see if `F3` has been pressed. When `F3` is pressed, BASIC branches to the subroutine beginning at Line 1000.

`KEY() STOP` temporarily halts trapping for the specified key. If the specified key is pressed, BASIC does not transfer program control to the `ON KEY() GOSUB` until you turn on key trapping again with `KEY() ON`. When you do turn on key trapping again, BASIC remembers that the key was pressed, and immediately branches to the subroutine.

`KEY() OFF` turns off key trapping. When key trapping is turned on again, BASIC does not remember that the key has been pressed.

Note: Key trapping only occurs while BASIC is running a program.

See `ON KEY() GOSUB` for more information on key trapping.

Sample Program

See `ON KEY() GOSUB`.

KILL

Statement

KILL *pathname*

Kills (deletes) *pathname* from disk.

Pathname is a standard file specification as described in Chapter 1.

You may delete any type of disk file. However, if the file is currently open, a `File already open` error occurs. You must close the file before deleting it.

Example

```
KILL "file.bas"
```

deletes File.bas if it exists in the current directory.

```
KILL "A:\REPORT\data"
```

deletes the file Data from the directory REPORT in Drive A.

LCOPY

Statement

LCOPY

Copies all text data on the screen to the printer.

Sample Program

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

This program segment prints exclamation points on the screen, and dumps them to the printer.

LEFT\$

Function

LEFT\$(string,number)

Returns the specified number of characters from the left portion of *string*.

Number must be an integer in the range 1 to 255. If *number* is equal to or greater than the length of the *string*, BASIC returns the entire string.

Examples

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

Since BIG FIERCE DOG is fewer than 20 characters, BASIC prints the whole phrase.

Sample Program

```
740 A$ = "TIMOTHY"  
750 B$ = LEFT$(A$, 3)  
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When you run this program, BASIC prints:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

Line 750 gets the 3 left characters of A\$ and stores them in B\$. Line 760 prints these 3 characters, a string, and the original contents of A\$.

LEN

Function

LEN(*string*)

Returns the number of characters in *string*. Blanks are counted.

Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE\$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

```
PRINT LEN("ORLANDO, FLORIDA")
```

prints 16.

LET

Statement

[LET] *variable* = *expression*

Assigns the value of *expression* to *variable*.

Variable is a numeric or string variable.

Expression is a numeric or string constant or expression. A BASIC function can be substituted for *expression*.

BASIC does not require assignment statements to begin with LET, but you might want to use LET to be compatible with versions of BASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"  
LET B1 = 1.23  
LET X = X - Z1  
LET X = SQR(B)
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant, expression, or function on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P  
560 LET P = 2001: PRINT "NOW P =" P
```

LINE/Graphics

Statement

LINE [[STEP](*x1,y1*)]-[STEP](*x2,y2*),[*color*][,B[F]]
[,*style*]

Draws a line or a box on the video display.

The STEP option tells BASIC that the (*x*,*y*) coordinates are relative to the last point referenced. If you use STEP with the second set of coordinates, the coordinates are relative to the first set of coordinates.

(*x1,y1*) specifies the point at which to begin the line. *x1* is the horizontal coordinate, and *y1* is the vertical coordinate. If you omit (*x1,y1*) BASIC begins the line at the last point referenced on the screen.

(*x2,y2*) specifies the point at which to end the line. *x2* is the horizontal coordinate and *y2* is the vertical coordinate.

Color indicates the color of the line.

See Chapter 8, "Displaying Text and Graphics" for information on coordinates and colors for the current screen mode.

If you specify coordinates that are not in the range of the current viewport, BASIC displays only that portion of the line that is within the viewport.

With the **B** option, BASIC draws a box. The points that you specify are opposite corners.

If you specify both the **B** and **F** options, BASIC draws a box and fills the box in with *color*.

Style lets you select the line-style used when drawing normal lines and unfilled boxes. *Style* is a 16-bit integer. Each bit represents a point in the line. If the bit equals 1, then the point is drawn. If the bit equals zero, then the point is not drawn. A zero bit does not erase a previously drawn point; therefore, you might want to draw a background line first to have a known background. The style pattern is repeated as necessary, to complete the line drawing.

Here are some sample styles showing the bit representation, the line drawn, and their hex equivalents:

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 equals &H5555
- - - - - - - - - is drawn

1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 equals &HDB6D
- - - - - - - - - is drawn

Examples

You can try these examples in the graphics modes. The color, size, and position of the image on the display vary, depending on the current screen mode.

LINE -(319, 199)

draws a line from the last point referenced to point 319,199 in the current color. This is the simplest form of the LINE statement. Note that when you omit the beginning points you must still include the hyphen.

LINE (0,0)-(319,199)

draws a diagonal line on the display in the current color.

LINE (0,100)-(319,100),1

draws a horizontal line across the display in Color 1.

LINE (0,0)-(100,100),,B

draws a box in the upper left corner of the display.

LINE (0,0)-(200,200),1,BF

draws a box on the display and fills it in with Color 1.

LINE (0,0)-(200,200),1,B, &H5555

draws a box outlined by a dashed line.

Sample Programs

```
10 CLS
20 LINE -(RND*319,RND*199),RND*4
30 GOTO 20
```

In the graphics modes, Lines 10-30 create a loop that draws random lines on the video display.

```
40 FOR X=0 TO 319
50 LINE (X,0)-(X,199),X AND 1
60 NEXT
```

In the graphics modes, Lines 40-60 draw an alternating pattern, turning on and off the line.

```
10 CLS
20 LINE -(RND*639,RND*199),RND*2,BF
30 GOTO 20
```

This program draws a random filled box in a graphics mode.

LINE INPUT

Statement

LINE INPUT[;][*“prompt”*]; *string variable*

Accepts an entire line (a maximum of 254 characters) from the keyboard. **LINE INPUT** is a convenient way to input string data without accidental entry of delimiters (commas, quotation marks, and so on.).

Prompt is a string constant enclosed in quotation marks that BASIC prints before waiting for input.

String variable is the variable to receive the input.

The only way to terminate the string input is to press ENTER. However, if **LINE INPUT** is immediately followed by a semicolon, pressing ENTER does not echo a carriage return to the display.

Note: You must place a space between **LINE** and **INPUT**.

LINE INPUT is similar to **INPUT**, except:

- BASIC does not display a question mark when waiting for input.
- Each **LINE INPUT** statement can assign a value to only 1 variable.
- Commas and quotation marks can be entered in the string input.
- Leading blanks are not ignored.

Examples

```
LINE INPUT A$
```

waits for input to A\$ without displaying a prompt.

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

displays the message and waits for input.

LINE INPUT#

Statement

LINE INPUT#*buffer, variable*

Accepts an entire line of data from a sequential file to a string *variable*.

Buffer is the number assigned to the file when you opened it.

This statement is useful when you want to read an ASCII format BASIC program file as data or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

- the end-of-file
- the 255th data character
- a carriage return

Other characters encountered—quotation marks, commas, leading blanks—are included in the string.

Note: You must place a space between LINE and INPUT#.

Example

If a ASCII format program file looks like this:

```
10 CLEAR 500
20 OPEN "I", 1, "prog"
```

then the statement:

```
LINE INPUT#1, A$
```

can be used repeatedly to read each program line, one at a time.

LIST**Statement****LIST** [*startline*][-[*endline*]][,*device*]

Lists a program in memory to the display.

Startline specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program.

If you omit both *startline* and *endline*, BASIC lists the entire program.

Device may be either SCRN: (screen) or LPT1: (line printer 1). If you omit *device*, the lines are listed to the screen.

You can temporarily stop the listing by pressing CTRL NUM LOCK. Press any key to continue the listing.

You can substitute a period (.) for either *startline* or *endline* to indicate the current line number.

Examples

```
LIST
```

displays the entire program.

```
LIST 50-85, "SCRN:"
```

displays lines in the range 50 to 85 on the screen.

```
LIST .-
```

displays the program line that you have entered or edited and all higher numbered lines on the screen.

```
LIST -227
```

displays all lines up to and including 227 on the screen.

```
LIST 227- , "LPT1:"
```

lists Line 227 and all higher numbered lines to the printer.

LLIST

Statement

LLIST [*startline*][-*endline*]

Lists program lines in memory to the printer.

Startline specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program.

If you omit both *startline* and *endline*, BASIC lists the entire program.

You can substitute a period (.) for either *startline* or *endline* to indicate the current line number.

LLIST assumes an 80-character-wide printer. You may change this by using the WIDTH statement with the LPRINT option.

Examples

```
LLIST
```

lists the entire program to the printer. To stop this process, press **CTRL** **NUM LOCK**. This causes a temporary halt in the computer's output to the printer. Press any key to continue printing.

```
LLIST 68-90
```

prints lines in the range 68 to 90.

LOAD

Statement

LOAD *pathname* [,R]

Loads a BASIC program from disk into memory.

Pathname is a standard file specification used to save the file to disk.

The R option tells BASIC to run the program. (LOAD with the R option is equivalent to the command RUN *pathname*.) When you specify the R option, BASIC leaves all open files open and runs the program automatically. If you omit the R option, BASIC wipes out any resident BASIC program, clears all variables, and closes all open files.

You can use LOAD inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-BASIC file, a `Direct statement in file error` occurs.

Example

```
LOAD "A:prog1.bas"
```

loads Prog1.bas from Drive A, and then returns to BASIC's prompt.

```
LOAD "prog1.bas"
```

loads Prog1.bas. Because no drive is specified, BASIC searches for Prog1.bas on the current drive.

LOC

Function

LOC(*buffer*)

Returns the current record position within a file.

Buffer is the number assigned to the file when you opened it.

You use LOC to determine the current record position, that is, the number of the last record processed since you opened the file.

When used with direct access files, LOC returns the record number accessed by the last GET or PUT statement.

When used with sequential files, LOC returns the number of 128-byte blocks that have been read or written.

Example

```
IF LOC(1)>55 THEN END
```

Program execution ends, if the current record position is greater than 55.

Sample Program

```
1310 A$ = "WILLIAM WILSON"  
1320 GET 1  
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD"  
LOC(1): CLOSE: END  
1340 GOTO 1320
```

This is a **portion** of a direct access program. Elsewhere the file has been opened and fielded. N\$ is a field variable. If N\$ matches A\$, the record number in which it was found is printed.

LOC/Communications

Function

LOC(*buffer*)

Returns the number of characters in the input queue if that number is 255 or less.

If the queue contains 256 or more characters, the LOC function returns the number 255. Since a string is limited to 255 characters, this limit eliminates the need for testing string size before reading data into the queue.

Buffer is the number assigned to the file when you opened it.

The default size for the input queue is 256 characters, but you can change the size by using the /C: option when loading BASIC.

Example

```
10 X=LOC(1)
20 If X>0 THEN A$=INPUT$(LOC(1),#1)
```

Line 10 checks to see if there are any characters in the input queue and stores the number of characters in the variable X. Line 20 tests the value of X. If X is greater than 0, there are characters in the input queue, and Line 20 returns the characters in the buffer into A\$.

Notice from the example that INPUT\$ is preferred over LINE INPUT# or INPUT# when reading communications files. This preference is because all ASCII characters might be significant in communications. INPUT\$ allows all characters to be read. The other statements do not. LINE INPUT# stops at a carriage return. INPUT# stops at a comma or a carriage return.

LOCATE

Statement

LOCATE [*row*][,*column*][,*cursor*][,*start*][,*stop*]]]

Positions the cursor on the screen.

Row is a numeric expression in the range 1 to 25 that indicates the screen row where you want to position the cursor.

Column is a numeric expression that indicates the screen column where you want to position the cursor. It may be in the range 1 to 40 or 1 to 80, depending on the current screen width.

Cursor indicates whether the cursor is visible or invisible. Set *cursor* to 1 for a visible cursor and to 0 for an invisible cursor.

Start specifies the starting scan line of the cursor. There are 7 scan lines available for the cursor. *Start* must be an integer in the range 0 to 31, where 0 is the top line and 31 is the bottom line.

Stop specifies the ending scan line of the cursor. *Stop* must be in the same range as *start*. If *stop* is less than *start* BASIC displays a split cursor.

Cursor, *start*, and *stop* are only effective in Screen Mode 0.

Examples

```
LOCATE 10,20,1,4
```

positions a half cursor on Row 10 in Column 20.

```
LOCATE 25,1,1,7
```

positions an underline cursor in the first position of the last line.

LOCK . . . UNLOCK

Statement

LOCK [#]*buffer* [,*record*]

UNLOCK [#]*buffer* [,*record*]

Controls access by other processes to all or part of an opened file.

LOCK and UNLOCK are used only by the compiler. Use them in a network environment where several users might work on the same file at the same time.

Buffer is the number used when you opened the file. The number sign is optional. It is provided for compatibility with other BASICs.

Record is the record or range of records to lock or unlock. It applies only to files opened for random input or output. If a file is opened for sequential input or output, LOCK and UNLOCK affect the entire file, regardless of the range specified.

To specify a range, use this format: *starting record* TO *ending record*. If you omit *starting record*, BASIC assumes you mean the first record in the file. If you omit *ending record*, BASIC assumes you mean the last record in the file. If you omit the *record* parameter completely, BASIC locks or unlocks the entire file.

Remove all locks with an UNLOCK statement before closing a file or stopping a program. Be sure the ranges in the corresponding LOCK and UNLOCK statements match exactly, as in this example:

```
LOCK 1, 1 TO 4
LOCK 1, 5 TO 8
UNLOCK 1, 1 TO 4
UNLOCK 1, 5 TO 8
```

The following statements do not match, and give unpredictable results:

```
LOCK 1, 1 TO 4
LOCK 1, 5 TO 8
UNLOCK 1, 1 TO 8
```

Examples

```
LOCK 2
```

locks all records in File 2.

```
LOCK 2, 32
```

locks Record 32 in File 2.

```
LOCK 2, 9 TO 32
```

locks all records in the range 9 to 32 in File 2.

```
LOCK 2, TO 32
```

locks all records in the range 1 to 32 in File 2.

Sample Program

```
1310 OPEN "MONITOR" AS 1 LEN = 59
1320 FIELD 1,15 AS PAYER$, 20 AS ADDRESS$, 20
AS PLACES$, 4 AS OWES$
1330 LET UPDATE$="Y"
1340 WHILE (UPDATE$="y" OR UPDATE$="Y")
1350   CLS:LOCATE 10,10
1360   INPUT "CUSTOMER NUMBER?   #"; NUMBER%
1370   LOCK 1, NUMBER%
1380   GET 1, NUMBER%
1390   LET DOLLARS! = CVS(OWES$)
1400   LOCATE 11,10: PRINT
"CUSTOMER: ";PAYER$
1410   LOCATE 12,10: PRINT
"ADDRESS:  ";ADDRESS$
1420   LOCATE 13,10: PRINT "CURRENTLY
OWES: $";DOLLARS!
1430   LOCATE 15,10: INPUT "CHANGE (+ OR -)";
CHANGE!
1440   LSET OWES$ = MKS$(DOLLARS!)
1450   PUT 1, NUMBER%
1460   UNLOCK 1, NUMBER%
1470   LOCATE 17,10: INPUT "UPDATE
ANOTHER? ";CONTINUE$
1480   LET UPDATE$=LEFT$(CONTINUE$,1)
1490 WEND
```

This program fragment opens a file and lets the user lock a record and update the information in it. When the user finishes, the program unlocks the record so that other people can use the file.

LOF

Function

LOF(*buffer*)

Returns the length of the file in bytes.

Buffer is the number assigned to the file when you opened it.

Example

```
Y = LOF(5)
```

assigns the length of the file in bytes to variable Y.

Sample Programs

During direct access to an existing file, you often need a way to know when you have read the last valid record. LOF provides a way:

```
1540 OPEN "R", 1, "unknown.txt", 128
1550 FIELD 1, 128 AS A$
1560 RCNUM% = 1 'START AT BEGINNING OF FILE
1570 RCSIZ% = 128 'SET RECORD SIZE
1580 IF RCNUM% * RCSIZ% > LOF(1) GOTO 1640
1590 'CHECK FOR END OF FILE
1600 GET 1, RCNUM% 'RECORD NUM. TO BE ACCESSED
1610 PRINT A$
1620 RCNUM% = RCNUM% + 1 'INCREMENT RECORD NUM
1630 GOTO 1580
1640 CLOSE
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an error.

These lines use LOF to determine where to start adding when you want to add to the end of a file:

```
1700 RCNUM% = (LOF(1) / RCSIZ%) + 1
1720 'HIGHEST EXISTING RECORD
1720 PUT 1, RCNUM% 'ADD NEXT RECORD
```

LOF/Communications	Function
--------------------	----------

LOF(*buffer*)

Returns the amount of free space in the input queue.

Buffer is the number assigned to the file when you opened it.

You can use LOF to determine when an input queue is getting full so that transmission is stopped.

The default length of the communications receive buffer is 256 bytes. If you wish, you can specify a different length by using the /C: switch when loading BASIC. (See “Options for Loading BASIC” in Chapter 2.)

LOG

Function

LOG(*number*)

Computes the natural logarithm of *number*.

Number must be greater than zero. LOG is the inverse of the EXP function.

BASIC always returns the result as a single precision number.

Examples

```
PRINT LOG(3.14159)
```

prints 1.144729.

```
Z = 10 * LOG(P5/P1)
```

performs the indicated calculation and assigns the value to Z.

Sample Program

This program demonstrates the use of LOG. It uses a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL (MILES)";  
D  
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F  
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))  
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE  
IS" L "DECIBELS."
```

LPOS

Function

LPOS(*number*)

Returns the logical position of the print head within the printer's buffer.

Number can be 0 or 1 to indicate LPT1:.

LPOS is only useful for checking the position of the print head after a LPRINT statement that is terminated by a semicolon to suppress the automatic carriage return. The statement containing LPOS is not executed until the LPRINT statement is finished printing.

LPOS does not necessarily give the physical position of the print head if the printed string contains the ASCII code for a carriage return. For example, if you are printing a string of 20 characters and the 10th character is the ASCII code for a carriage return, the printer advances to the next line after printing the 9th character. Then it prints the remaining 10 characters. If the string is terminated by a semicolon to suppress the automatic line feed, the physical location of the print head is at position 10. LPOS returns a value of 21, however, because that is the logical location of the print head.

Example

You may want to use LPOS to determine whether there is enough room to continue printing more variables on the same line.

```
100 IF LPOS(X)>60 THEN LPRINT
```

If the printer has printed more than 60 characters, a carriage return is sent so that the printer skips to the next line.

LPRINT

Statement

LPRINT [USING *format*;] *data*[,*data*,...]

Prints *data* on the printer.

LPRINT assumes a print width of 80 characters. You may change the width using the WIDTH statement with the LPRINT option.

See PRINT and PRINT USING for more information on formatting the output.

Examples

```
LPRINT (A * 2)/3
```

prints the value of expression $(A * 2)/3$ on the printer.

```
LPRINT TAB(50) "TABBED 50"
```

moves the printer carriage to tab position 50 and prints TABBED 50. (Refer to the TAB function.)

```
LPRINT USING "#####.##"; 2.17
```

sends the formatted value `bbbb2.2` to the printer.

LSET

Statement

LSET *field name* = *data*

Moves *data* to the direct access buffer and places it in *field name*, in preparation for a PUT statement.

Field name is a string variable defined in a FIELD statement.

You must have used FIELD to set up buffer fields before using LSET.

You must convert numeric values to string values before they are LSET. See MKI\$, MKD\$, MKS\$.

You use LSET to left-justify the variable in the field. If the field is larger than the variable it is receiving, the field is filled with blanks on the right. If the variable is larger than the field, characters are truncated on the right. The complementary command of LSET is RSET.

See also Chapter 7, and OPEN, CLOSE, FIELD, GET, PUT, and RSET.

Example

Suppose NM\$ and AD\$ have been defined as field names for a direct access file buffer. NM\$ has a length of 18 characters; AD\$ has a length of 25 characters. The statements:

```
LSET NM$ = "JIM CRICKET, JR."  
LSET AD$ = "2000 EAST PECAN ST."
```

set the data in the buffer as follows:

```
JIM    CRICKET, JR.      2000    EAST    PECAN    ST.      
```

Notice that filler blanks are placed to the right of the data strings in both cases. If we use RSET statements instead of LSET, the filler spaces are placed to the left. This is the only difference between LSET and RSET.

MERGE

Statement

MERGE *pathname*

Loads a BASIC program and merges it with the program currently in memory.

Pathname is a standard file specification as described in Chapter 1. The filename is required. The file must be in ASCII format; that is, it must have been saved with the A option.

Program lines in *pathname* are inserted into the resident program in sequential order. For example, suppose that 3 lines from *pathname* are numbered 75, 85, and 90, and 3 lines from the resident program are numbered 70, 80, and 90. When you use MERGE on the 2 programs, this portion of the merged program is now numbered 70, 75, 80, 85, 90.

If line numbers on the new program coincide with line numbers in the resident program, the new program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, BASIC returns its prompt.

Example

Suppose you have a BASIC program on disk, Prog2.txt (saved in ASCII), that you want to merge with the program you have in memory:

```
MERGE "prog2.txt"
```

merges the 2 programs.

Sample Programs

MERGE provides a convenient means of putting program modules together. For example, an often-used set of BASIC subroutines can accompany a variety of programs with this command.

Suppose the following program is in memory:

```
80 REM          MAIN PROGRAM
90 REM LINE NUMBER RESERVED FOR SUBROUTINE HOOK
100 REM         PROGRAM LINE
110 REM         PROGRAM LINE
120 REM         PROGRAM LINE
130 END
```

And suppose the following subroutine, Sub.txt, is stored on disk in ASCII format:

```
90 GOSUB 1000 `SUBROUTINE HOOK
1000 REM      BEGINNING OF SUBROUTINE
1010 REM      SUBROUTINE LINE
1020 REM      SUBROUTINE LINE
1030 REM      SUBROUTINE LINE
1040 RETURN
```

You can MERGE the subroutine with the main program with:

```
MERGE "sub.txt"
```

The new program in memory is:

```
80 REM          MAIN PROGRAM
90 GOSUB 1000 `SUBROUTINE HOOK
100 REM         PROGRAM LINE
110 REM         PROGRAM LINE
120 REM         PROGRAM LINE
130 END
1000 REM        BEGINNING OF SUBROUTINE
1010 REM        SUBROUTINE LINE
1020 REM        SUBROUTINE LINE
1030 REM        SUBROUTINE LINE
1040 RETURN
```


MID\$**Statement**

MID\$(oldstring,start[,length]) = newstring

Replaces a portion of *oldstring* with *newstring*.

Oldstring is the variable name of the string you want to change.

Start is a number specifying the position of the first character you want to change.

Length is a number specifying the number of characters you want to replace.

Newstring is the string to replace a portion of *oldstring*.

The length of the resultant string is always the same as the original string. If *newstring* is shorter than *length*, the entire replacement string is used.

Examples:

```
10 A$ = "LINCOLN"  
20 MID$(A$,3,4) = "12345": PRINT A$
```

prints LI1234N.

Replace Line 20 with:

```
20 MID$(A$,5) = "01": PRINT A$
```

and BASIC prints LINC01N.

MID\$

Function

MID\$(string, start [,length])

Returns a substring of a string.

Length is the number of characters in the substring. It must be in the range 1 to 255.

Start specifies the position in the string from which to get the substring.

If you omit *length* or if there are fewer than that number of characters to the right of *start* position, BASIC returns all characters to the right of the character at the *start* position including that character at *start*.

If *start* is greater than *number* of characters in *string*, BASIC returns a null string.

Examples

```
10 A$ = "WEATHERFORD"  
20 PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F\$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER (NNN-NNN-NNNN)";  
PH$  
210 EX$ = MID$(PH$, 5, 3)  
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first 3 digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last 4 digits) and picks out the exchange.

MKDIR

Statement

MKDIR *pathname*

Creates the directory specified by *pathname*.

Pathname is a standard directory specification as described in Chapter 1. If you omit the drive identifier, the directory is created on the current drive. If you omit the root directory symbol (\), the directory is created in the current directory.

Examples

```
MKDIR "A:\ACCTS\PAYABLE"
```

creates the directory PAYABLE in the ACCTS directory on Drive A.

```
MKDIR "\\ADDRESS"
```

creates the directory ADDRESS in the root directory on the current drive.

```
MKDIR "NAMES"
```

creates the directory NAMES in the current directory on the current drive.

MKD\$, MKI\$, MKS\$	Function
----------------------------	-----------------

MKD\$(*double precision expression*)

MKI\$(*integer expression*)

MKS\$(*single precision expression*)

Converts numeric values to string values.

Any numeric value that is placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These 3 functions are the inverse of CVD, CVI, and CVS. The byte values that make up the number are not changed; only 1 byte, the internal data-type specifier, is changed so that numeric data can be placed in a string variable.

MKD\$ returns an 8-byte string; MKI\$ returns a 2-byte string; and MKS\$ returns a 4-byte string.

Example

```
LSET AVG$ = MKS$(0.123)
```

Sample Program

```
1350 OPEN "R", 1, "test.dat", 14
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKS$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1, 1
1410 CLOSE 1
```

For a program that retrieves the data from Test.dat, see CVD/CVI/CVS.

NAME

Statement

NAME *old filename* AS *new filename*

Renames *old filename* as *new filename*.

With this statement, the data in the file is left unchanged. The *new filename* may not contain a path, password, or drive specification.

You can only rename a file in the current directory.

Example

```
NAME "file.bas" AS "file.old"
```

BASIC renames File.bas as File.old.

NEW

Statement

NEW

Deletes the program currently in memory and clears all variables. NEW also closes all open files, turns off the trace function and resets the music background.

Example

```
NEW
```

OCT\$

Function

OCT\$(*number*)

Returns the octal value of *number*.

OCT\$ returns a string that represents the octal value of a decimal *number*. The value returned is like any other string—it cannot be used in a numeric expression.

Examples

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the strings 36, 62, and 132.

```
Y$ = OCT$(X/84)
```

Y\$ is a string representation of the integer quotient X/84 to base 8.

ON COM() GOSUB Statement

ON COM(*channel*) GOSUB *line*

Transfers program control to a subroutine beginning at *line* when any character is received on the specified communications channel.

Channel selects Communications Channel 1 or 2.

Line is the first line of the subroutine to be executed when activity occurs on the specified communications channel. If you specify Line 0, you turn off communications trapping. Use the RETURN statement to exit the subroutine.

BASIC executes the ON COM() GOSUB statement only if a COM() ON statement has been previously executed to enable communications trapping. If a COM() STOP statement has been issued to halt communications trapping temporarily, BASIC executes the subroutine immediately after the next COM() ON statement.

When you execute the ON COM() GOSUB statement, BASIC immediately issues a COM() STOP statement to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another COM() ON statement to enable communications trapping again, unless the subroutine executes a COM() OFF statement.

Example

```
10 OPEN "COM1:" AS 1
20 COM(1) ON
30 ON COM(1) GOSUB 100
40 GOTO 40
.
.
.
100 A$=INPUT$(1,1):PRINT A$;
110 RETURN
```

Line 20 turns on communications trapping on Channel 1. After executing each program statement, BASIC checks to see if any character has come into the communication channel's buffer. If any has, BASIC immediately executes the subroutine beginning

at Line 100. The communications trap subroutine reads the received character and displays it on the screen.

To avoid the overhead of trapping at high baud rates, we recommend that the communications trap subroutine read the entire message from the buffer.

ON ERROR GOTO

Statement

ON ERROR GOTO *line*

Transfers control to *line* if an error occurs.

This lets your program recover from an error and continue execution. (Normally, you have a particular type of error in mind when you use the ON ERROR GOTO statement.)

You must execute an ON ERROR GOTO before the error occurs.

To disable it, execute an ON ERROR GOTO 0, which causes BASIC to stop execution and print an error message. This is recommended for errors that are trapped and from which you cannot recover.

Note: If an error occurs during execution of an error-handling routine, that error message is printed and execution terminates. Error trapping does not occur within the error handling routine.

The error-handling routine must be terminated by a RESUME statement.

Example

```
10 ON ERROR GOTO 1500
```

branches program control to Line 1500 if an error occurs anywhere after Line 10.

Sample Program

See ERROR.

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

Looks at n and transfers program control to the subroutine indicated by the n th line listed.

For example, if n equals 1, BASIC branches to the first line listed; if n equals 2, BASIC branches to the second line listed.

Line is the subroutine line at which execution begins when BASIC makes the branch.

N must be a number in the range 0 to 255. If necessary, BASIC rounds n to an integer before evaluating it. If n is 0 or greater than the number of line numbers listed, BASIC continues with the next statement. If n is negative or is greater than 255, an Illegal function call error occurs.

Use the RETURN statement to exit the subroutine.

Example

```
10 ON Y GOSUB 1000, 2000, 3000
```

If Y equals 1, BASIC branches to a subroutine, beginning at Line 1000. If Y equals 2, BASIC branches to a subroutine, beginning at Line 2000. If Y equals 3, BASIC branches to a subroutine, beginning at Line 3000.

If Y is outside the range 1 to 3, BASIC either continues with the next statement or generates an Illegal function call, as mentioned earlier.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

ON/GOTO

Statement

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

Looks at n and transfers program control to the n th line listed.

For example, if n equals 1, BASIC branches to the first line listed; if n equals 2, BASIC branches to the second line listed.

N must be in the range 0 to 255. If necessary, BASIC rounds n to an integer before evaluating it. If n is 0 or is greater than the number of line numbers listed, BASIC continues with the next statement. If n is negative or is greater than 255, an illegal function call error occurs.

Example

```
10 ON MI GOTO 150, 160, 170, 150, 180
```

tells BASIC to evaluate MI. If MI equals 1, BASIC branches to Line 150; if MI equals 2, BASIC branches to Line 160; and so on. If MI is outside of the range 1 to 5, BASIC either continues with the next statement or generates an illegal function call, as mentioned earlier.

Sample Program

```
5 REM <CAPS> MUST BE ON
10 INPUT "ENTER A,B, or C,";L$
20 L=ASC (L$)
30 ON L-64 GOTO 50, 60, 70
40 PRINT "TRY AGAIN":GOTO 10
50 PRINT "YOU TYPED 'A'":END
60 PRINT "YOU TYPED 'B'":END
70 PRINT "YOU TYPED 'C'":END
```


ON KEY() GOSUB

Statement

ON KEY(*number*) GOSUB *line*

Transfers program control to a subroutine when you press the specified key.

Number may be a number in the range 1 to 20, indicating the number of the key to trap. Function keys use their corresponding function key numbers. The cursor direction keys are numbered:

	11
	12
	13
	14

User keys are numbered 15 through 20. User keys are defined with the KEY statement.

Line is the first line number in the subroutine to execute when the specific key is pressed. If you specify Line 0, you turn off key trapping for that key. It is the same as executing a KEY() OFF statement. Use the RETURN statement to exit the subroutine.

BASIC executes the ON KEY() GOSUB statement only if a KEY() ON statement has been executed previously to enable key trapping for that key.

If a KEY() STOP statement has been issued to halt key trapping for that key temporarily, BASIC executes the subroutine immediately after the next KEY() ON statement for that key.

When you execute the ON KEY() GOSUB statement, BASIC immediately issues a KEY() STOP statement for that key to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another KEY() ON statement for that key to enable key trapping again, unless the subroutine executes a KEY() OFF statement for that key.

Sample Program

```
10 KEY(5) ON
20 KEY(1) ON
30 ON KEY(5) GOSUB 80
40 ON KEY(1) GOSUB 100
50 FOR I = 1 TO 100
60 PRINT "NO KEY PRESSED"
70 NEXT I:END
80 PRINT "KEY(5) PRESSED *****"
90 RETURN
100 PRINT "KEY(1) PRESSED ;;;;;;;;;;"
110 RETURN
```

This program sets up **F5** and **F1** to be trapped.

ON PEN GOSUB

Statement

ON PEN GOSUB *line*

Transfers program control to a subroutine when you activate the light pen.

Line is the first line number in the subroutine to execute when the light pen is activated. If you specify Line 0, you turn off trapping for the pen. It is the same as executing a PEN OFF statement. Use the RETURN statement to exit the subroutine.

BASIC executes the ON PEN GOSUB statement only if a PEN ON statement has been executed previously to enable light pen trapping.

If a PEN STOP statement has been issued to halt trapping for the pen temporarily, BASIC executes the subroutine immediately after the next PEN ON statement.

When you execute the ON PEN GOSUB statement, BASIC immediately issues a PEN STOP statement to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another PEN ON statement to enable pen trapping again, unless the subroutine executes a PEN OFF statement.

Example

```
10 PEN ON
20 ON PEN GOSUB 1000
30 REM
.
.
.
500 END
1000 REM PROCESSING ROUTINE
.
.
.
1100 RETURN 30
```

Line 10 turns on pen trapping. After each program statement is executed, BASIC checks to see if the pen has been activated. If it has, BASIC immediately executes the subroutine at Line 1000.

ON PLAY() GOSUB

Statement

ON PLAY(*number*) GOSUB *line*

Transfers program control to a subroutine when the number of notes in the background music buffer goes from *number* to *number* minus 1. This event trapping allows continuous music by letting you maintain a full music buffer.

Number is an integer in the range 1 to 32.

Line is the number of first line of the subroutine to execute. If you specify Line 0, you turn off play event trapping. Use the RETURN statement to exit the subroutine.

BASIC executes the ON PLAY() GOSUB statement only when playing background music (PLAY "MB") and if the PLAY ON statement has been executed to enable event trapping.

If a PLAY STOP statement has been issued to halt event trapping temporarily, BASIC executes the subroutine immediately after the next PLAY ON statement.

When you execute the ON PLAY() GOSUB statement, BASIC immediately issues a PLAY() STOP to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another PLAY() ON statement to enable trapping again, unless the subroutine executes a PLAY() OFF statement.

Notes: BASIC does not issue a play event trap if the background music queue is already empty when you execute a PLAY ON.

The PLAY statement is supported by a 32-element music queue. Given that "normal" and "staccato" notes are constructed from 2-note elements, the queue can contain as few as 16 notes or as many as 32 notes.

Therefore, select conservative values for the trap number. For example, if *number* is set at 32, event traps might happen so often that there is little time to execute the rest of your program. It is suggested that the trap number be less than 16 for better performance.

Example

```
100 PLAY ON
.
.
540 PLAY "MB L1 XZITHER$"
550 ON PLAY(8) GOSUB 6000
.
.
.
6000 REM **BACKGROUND MUSIC**
6010 LET COUNT% = COUNT% + 1
.
.
6999 RETURN
```

Control branches to a subroutine when the background music buffer decreases to 7 notes.

ON STRIG() GOSUB

Statement

ON STRIG(*number*) GOSUB *line*

Transfers program control to a subroutine when you press one of the joystick's buttons.

Number specifies the number of the button pressed and is one of the following:

0	left joystick, button 1
2	right joystick, button 1
4	left joystick, button 2
6	right joystick, button 2

Line is the first line number of the subroutine to be executed when you press one of the joystick's buttons. If you specify Line 0, you turn off trapping for the joysticks. Use RETURN to exit the subroutine.

BASIC executes the ON STRIG() GOSUB statement only if a STRIG ON statement has been executed previously to enable joystick trapping.

If a STRIG STOP statement has been issued to halt joystick trapping temporarily, BASIC executes the subroutine immediately after the next STRIG ON statement.

When the ON STRIG() GOSUB statement is executed, BASIC immediately issues a STRIG STOP statement to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another STRIG ON statement to enable joystick button trapping again, unless the subroutine executes a STRIG OFF statement.

Sample Program

```
5 STRIG(0) ON: STRIG(2) ON
10 ON STRIG(0) GOSUB 1000
20 ON STRIG(2) GOSUB 2000
30 PRINT "Press one of the joystick buttons."
40 FOR I = 1 TO 3000:NEXT I
50 GOTO 30
1000 PRINT "You pressed the left button."
:RETURN
2000 PRINT "You pressed the right button."
:RETURN
```

Lines 10 and 20 turn on joystick trapping. Line 30 instructs you to press one of the buttons. Line 40 waits for you to press a button. If you press the left button, BASIC transfers program control to the subroutine at Line 1000. If you press the right button, BASIC transfers program control to the subroutine at Line 2000. If you do not press a button, Line 50 returns to print the message again. This program is a continuous loop. To end the program, press **CTRL****BREAK**.

ON TIMER() GOSUB

Statement

ON TIMER(*number*) GOSUB *line*

Transfers program control to a subroutine when the specified period of time has elapsed.

Number indicates the number of seconds. *Number* may be a value in the range 1 to 86400 (86400 seconds = 24 hours).

Line is the first line number in the subroutine to execute when the specified time has passed. If you specify Line 0, you turn off trapping for the timer. Use RETURN to exit the subroutine.

BASIC executes the ON TIMER() GOSUB statement only if a TIMER ON statement has been executed previously to enable time event trapping.

If a TIMER STOP statement has been issued to halt time event trapping temporarily, BASIC executes the subroutine immediately after the next TIMER ON statement.

When you execute the ON TIMER() GOSUB statement, BASIC immediately issues a TIMER STOP to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another TIMER ON statement to enable trapping again, unless the subroutine executes a TIMER OFF statement.

Example

```
10 TIMER ON
20 ON TIMER (60) GOSUB 1000
30 REM
.
.
.
500 END
.
.
.
1000 REM PROCESSING ROUTINE
.
.
.
1100 RETURN 30
```

Line 10 turns on timer trapping. After each statement is executed, BASIC checks to see if the specified time has elapsed. If it has, BASIC immediately executes the subroutine at Line 1000.

OPEN

Statement

OPEN *mode*,[#]*buffer*,[*pathname*][*dev:*][*record length*]

OPEN [*pathname*][*dev:*] [**FOR** *mode*] [*access*] **AS** [#]*buffer* [**LEN** = *record length*]

Establishes an input/output path for a file or device.

Buffer is an integer in the range 1 to 255. It specifies the I/O buffer in memory to use when accessing the file. The number sign (#) is optional. It is provided for compatibility with other BASICs.

Pathname is a standard file specification as described in Chapter 1. If you omit *pathname*, you must include *dev:*.

dev: specifies the device to be opened for communication.

Record length is an integer in the range 1 to 32768 that sets the record length for direct access files. If you omit *record length*, BASIC assumes a default record length of 128 bytes, unless you used /I and /R when loading BASIC. **Do not** use this option with sequential access files.

Mode specifies any of the following:

O or OUTPUT	sequential output mode
I or INPUT	sequential input mode
A or APPEND	sequential extension of an existing file
R or RANDOM	direct input/output mode

In the first form of the syntax, you must use the abbreviated form of *mode*, and it must be enclosed in quotation marks.

In the second form of the syntax, you must specify the complete word for *mode*. You may not specify RANDOM. If you want to use direct access in the second form of the syntax, omit *mode*.

You may open a file for output in only one buffer at a time. Once you assign a buffer to a file with the OPEN statement, you cannot use that buffer in another OPEN statement until you close the first file. However, BASIC lets you access the same file for input by opening it in different buffers. You may keep several records from the same file in memory for quick access.

If you try to open a non-existing file for input, BASIC returns a `File not found` error.

If you try to open a non-existing file for output, BASIC creates the file.

If you try to open a non-existing file for append, BASIC creates the file and sets the mode to `RANDOM`.

If you try to open a file for direct access with a record length that does not match the record length assigned to the file when it was created, an error occurs.

Access supports networking and controls the processes that can access the file and the degree to which they can do so. *Access* can be any of the following:

SHARED	Any process on any machine can read from or write to the file.
LOCK READ	Only the current process can read the file. (The system grants LOCK READ only if no other process already has LOCK READ access to the file.)
LOCK WRITE	Only the current process can write to the file. (The system grants LOCK WRITE only if no other process already has LOCK WRITE access to the file.)
LOCK READ WRITE	Only the current process can read from or write to the file. (The system grants LOCK READ WRITE only if no other process already has LOCK READ, LOCK WRITE, or LOCK READ WRITE access to the file.)

If you omit *access*, the current process can open the file any number of times for reading and writing, but all other processes are denied access.

Remarks

You can refer to the same file in a subdirectory by different pathnames. For example, if MARY is your current directory, then:

```
OPEN "REPORT"  
OPEN "ES\MARY\REPORT"  
OPEN "..\MARY\REPORT"  
OPEN "...\MARY\REPORT"
```

all refer to the same file.

It is nearly impossible for BASIC to know that the file is the same simply by looking at the path. For this reason, BASIC does not let you open a file for output or append if it is on the same disk, even if the path is different.

BASIC Devices

The BASIC devices are:

```
KYBD:    LPT:n  
SCRN:    CON:  
COMn:
```

The BASIC file I/O system lets you take advantage of user-installed devices. (See your *MS-DOS Reference Manual* for information on character devices.)

You can open and use character devices in the same manner as disk files. However, BASIC does not buffer the characters the same as for disk files. Instead, it sets the record length to one.

At the end of a line, BASIC sends only a carriage return (X'0D'). If the device requires a line feed (X'0A'), the device driver must provide it. When writing device drivers, keep in mind that other BASIC users will want to read and write control information. Use the BASIC IOCTL statement and the IOCTL\$ function to handle writing and reading of device control data.

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

Examples

```
OPEN "R", 2, "test.dat"
```

opens the file Test.dat in direct access mode, using Buffer 2. If Test.dat does not exist, BASIC creates it on the current drive. The record length is 128 bytes.

```
OPEN "LPT1:" FOR OUTPUT AS 2
```

opens the printer for sequential output using Buffer 2.

```
OPEN "A:\PAYROLL\data.bas" FOR INPUT AS 1
```

opens the file Data.bas in the PAYROLL directory on Drive A for sequential input using Buffer 1.

```
OPEN "\DEV\CON:" FOR OUTPUT AS 1
```

opens the console for sequential output using Buffer 1.

```
OPEN "mailing.dat" FOR APPEND AS 1
```

opens the file Mailing.dat, using Buffer 1, and allows you to add data without destroying the current contents of the file.

OPEN/Communications Statement

OPEN "**COM***channel*: [*speed*] [,*parity*] [,*data*][,*stop*]
[,*RS*][,*CS*[*seconds*]][,*DS*[*seconds*]][,*CD*[*seconds*]][,*mode*]
[,*PE*] [,*LF*]" [**FOR** *mode*] **AS** [#]*buffer*[*LEN* = *number*]

Opens a file and allocates a buffer for RS-232C (Asynchronous Communications Adapter) communication.

Channel can be 1 or 2 to select the communications channel to be opened.

Speed is an integer specifying the transmit and receive rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 2400, 4800, and 9600. If you omit *speed*, BASIC sets the *speed* at 300 bps.

Parity is a constant specifying the parity to be used when the data is transmitted and received. The constant must be one of the following:

- | | |
|---|--|
| E | EVEN transmit parity, EVEN receive parity checking. |
| O | ODD transmit parity, ODD receive parity checking. |
| M | parity bit always transmitted and received as a mark (1 bit). |
| S | parity bit always transmitted and received as a space (0 bit). |
| N | no transmit parity, no receive parity checking. |

If you omit *parity*, BASIC assumes E (EVEN).

Data is an integer specifying the number of transmit and receive bits. Valid values are 5, 6, 7, and 8. If you do not specify *data*, BASIC assumes 7.

Note: Eight data bits **with** parity is illegal. Therefore, if you specify 8 data bits, you must specify parity N.

Stop must be either 1 or 2 to indicate the number of stop bits. If you omit *stop*, 75 and 110 bps transmit 2 *stop* bits, and all other

speeds transmit 1 *stop* bit. However, if you specify 5 as the *data*, the 2 stop bits actually mean 1½ stop bits.

Mode is one of the following string expressions:

OUTPUT	sequential output mode
INPUT	sequential input mode

If you omit *mode*, BASIC assumes it to be random input/output.

Buffer is a number in the range 1 to 15, indicating the buffer that accesses the file. The number sign (#) is optional. It is provided for compatibility with other BASICs.

Number specifies the maximum number of bytes that can be accessed in the communications buffer by GET and PUT statements. If you omit the LEN option, BASIC assumes 128 bytes.

The parameters *speed*, *parity*, *data*, and *stop* are all positional. That is, they must be in the order specified in the syntax. If you omit one of the parameters, you must still include the comma to hold its position.

The remaining parameters are not positional. They may be in any order, or you may omit them. They control the software communication signal lines between 2 terminals. If you omit the CS, DS, or CD options, the signals are not checked at all. Include them only if you are testing these software signals.

The RS option suppresses the Request To Send (RTS) signal. Request To Send is a signal that is sent from the sending terminal to the receiving terminal to ensure that the receiving terminal is ready to accept communication data. When you execute an OPEN COM statement, the RTS line is turned on, unless you include the RS option.

The CS option controls the Clear To Send (CTS) signal which is sent from the receiving terminal to the sending terminal to let the sending terminal know that the receiving terminal is ready to receive.

You can think of RTS and CTS as a hand-shaking exercise, in which the 2 terminals let each other know that they are ready to send and/or receive data. RTS is an output signal from the sending terminal, and CTS is an input signal to the sending terminal.

The DS option controls the Data Set Ready (DSR) signal. The DSR signal ensures that a data set, such as a modem, is present to transmit the data.

The CD option controls the Carrier Detect (CD) signal. The CD signal is an input signal which ensures that the data set is ready to transmit the data.

The *seconds* argument in the CS, DS, and CD options specifies the number of milliseconds to wait for the signal before returning a Device Timeout error. *Seconds* may be in the range 0 to 65535. If you omit *seconds* or specify a zero, the signal is not checked.

If you specify RS, *seconds* defaults to zero for CS. If you omit RS, the default for CS is 1000. Either an RS or a CS is required. That is, if you omit RS, the Clear To Send signal is not checked. If you include RS, OPEN COM waits 1 second for CS before issuing a Device Timeout error.

If you omit *seconds* after the DS option, the default value is 1000, and OPEN COM waits 1 second before issuing a Device Timeout error. If you omit *seconds* after CD, the default is zero and the signal is not checked.

I/O statements to a communications file do not execute if these signals are off. The system waits 1 second before returning a Device Timeout error. Specifying these options lets you ignore these signals or specify the length of time to wait for the signal.

The LF option sends a line feed character after every carriage return. This is useful if you are printing the communication data to a serial line printer. A line feed is also sent after the carriage return that is the result of the width setting. Note that when you specify the LF option, INPUT# and LINE INPUT# stop when they read a carriage return and ignore the line feed.

Mode specifies the type of data that is transmitted. It may be either BIN for binary mode or ASC for ASCII mode. If you omit *mode*, OPEN COM1 opens the device in binary mode.

If you specify the BIN mode, OPEN COM does not expand tabs to spaces, does not force a carriage return at the end of the line, does not recognize Control Z as an end-of-file, and ignores the LF option.

If you specify the ASC mode, OPEN COM expands tabs to spaces, forces a carriage return at the end of the line, and recognizes Control Z as the end-of-file. When you close the channel, Control Z is sent over the RS-232 line.

The PE option enables parity checking. With the option on, parity errors cause Device I/O errors and turn on the high order bit for 7 or fewer data bits. The default is no parity checking. Framing and overrun errors always cause Device I/O errors and turn on the high order bit, regardless of whether or not you use the PE option.

Examples

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

opens Buffer 1 for Communications Channel 1 at a rate of 300 bps with even parity, 7 data bits, and 1 stop bit. RTS is sent.

```
OPEN "COM2:9600,N,8,1,BIN" AS 2
```

opens Buffer 2 for Communications Channel 2 at a rate of 9600 bps with no parity, 8 data bits, and 1 stop bit. The data is binary.

```
OPEN "COM1: 4800,,,CS3000,DS2000" AS 1
```

opens Buffer 1 for Communications Channel 1 at a rate of 4800 bps with even parity, 7 data bits, and 1 stop bit. RTS is sent. OPEN COM issues a Device Timeout error if there is no CS signal after 3 seconds and no DS signal after 2 seconds. Note that even though parity, data, and stop are not included, the commas are required.

OPTION BASE

Statement

OPTION BASE *value*

Sets *value* as the minimum value for an array subscript.

Value may be 1 or 0. The default is 0.

If you use this statement in a program, it must precede the DIM statement.

If the statement:

```
OPTION BASE 1
```

is executed, 1 is the lowest value an array subscript may have.

OUT

Statement

OUT *port, data byte*

Sends a *data byte* to a machine output *port*. A port is an input/output location in memory.

Port is an integer in the range 0 to 65535.

Data byte is an integer in the range 0 to 255.

Example

```
OUT 32,100
```

sends 100 to port 32.

PAINT/Graphics

Statement

PAINT (*x,y*) [,*color*][,*border*][,*background*]]

Fills in an area on the display with a selected color or pattern.

(*x,y*) specify the coordinates where the painting begins. *x* is the horizontal coordinate, and *y* is the vertical coordinate.

Color can be either a number or a string expression. If *color* is a number it specifies a color number available in the current screen mode.

If *color* is a string expression, it specifies the mask to be used for tiling. The tiling mask describes a pattern to be used when painting and is in the form:

CHR\$(&Hnn) + CHR\$(&Hnn) + CHR\$(&Hnn)...

Border specifies the border color at which to stop painting, and must be a color number in the current palette. If you omit *border*, BASIC assumes the value of *color*.

See Chapter 8, "Displaying Text and Graphics," for information on coordinates and colors for the current screen mode.

Background is a 1-byte string expression specifying which color to skip when checking for borders while paint tiling.

BASIC begins to change the color of pixels at the point you specify with *x* and *y* coordinates. BASIC continues to change the color of every pixel that is not the same color as *color*. When BASIC paints 1 line of pixels without changing the color of any pixel in that line, PAINT is complete.

However, you may continue past this point while tile painting. The background option tells PAINT what background tile pattern or color byte to skip when checking for the boundary.

This means that instead of stopping when 1 line of points has been painted without changing the color, PAINT can continue, if you specify *background*. For example, normally you cannot draw alternating blue and red lines on a red background because PAINT stops after painting the first red line. However, by specifying red as the background color (&HAA), you can draw the red line over the red background.

PAINT must start on a nonborder point. If the point is already *border* or *color* color, BASIC does not execute the PAINT statement.

PAINT can fill any figure, but painting jagged edges or very complex figures may result in an Out of memory error. If this happens, you must use the CLEAR statement to increase the amount of stack space available.

Tiling

Tiling lets you select a pattern to be used when painting an area on the screen. The tile mask is 8-bits wide and may be a maximum of 64 bytes long:

x,y	8	7	6	5	4	3	2	1	tile byte
0,0	1
0,1	2
0,2	3
0,3	4
.					.				.
.					.				.
.					.				.
0,63	64

Each byte in the mask represents 8 points along the horizontal row and 1 point along the vertical row. PAINT repeats the tile mask pattern (horizontally and vertically) to create a uniform pattern over the entire area being painted.

In high resolution graphics, 1 bit of the tile mask equals 1 point on the screen. Therefore, each position in the tile mask with the bit value one (1) is drawn. You can paint a pattern of Xs with this tile mask:

byte	8	7	6	5	4	3	2	1	
0	1	0	0	0	0	0	0	1	CHR\$(&H81)
1	0	1	0	0	0	0	1	0	CHR\$(&H42)
2	0	0	1	0	0	1	0	0	CHR\$(&H24)
3	0	0	0	1	1	0	0	0	CHR\$(&H18)
4	0	0	0	1	1	0	0	0	CHR\$(&H18)
5	0	0	1	0	0	1	0	0	CHR\$(&H24)
6	0	1	0	0	0	0	1	0	CHR\$(&H42)
7	1	0	0	0	0	0	0	1	CHR\$(&H81)

In 4-color graphics, 2 bits correspond to each point on the screen. That is, each byte of the tile mask describes only 4 points. These 2 bits describe the color for the point being drawn. The following chart shows the values for the given colors. Remember, Color 0 is the set background color. (See COLOR.)

Palette 0	Palette 1	binary value
green	cyan	01
red	magenta	10
brown	high-intensity white	11

The following tile mask sets up a star pattern in green and brown using Palette 0 or in cyan and high intensity white using Palette 1.

BYTE

0	01 00 00 01	CHR\$(&H41)
1	00 01 01 00	CHR\$(&H14)
2	11 11 11 11	CHR\$(&HFF)
3	00 01 01 00	CHR\$(&H14)
4	01 00 00 01	CHR\$(&H41)

PEEK

Function

PEEK(*memory location*)

Returns a byte from *memory location* .

Memory location must be in the range -32768 to 65535.

The value returned is an integer in the range 0 to 255. (For the interpretation of a negative value of *memory location*, see the VARPTR statement.)

PEEK is the complementary function of the POKE statement.

Example

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

BASIC returns the value stored at address 5A00 and stores it in variable A.

PEN

Function

PEN(*number*)

Returns the light pen's coordinates.

Number is a number in the range 0 to 9 that tells BASIC what to return. Values 0-5 return x,y coordinates corresponding to the current screen mode. Values 6-9 return the character row or column position.

- 0 Returns a -1 if pen button has been pressed since the last poll. Returns a 0 if not.
- 1 Returns the x-coordinate (horizontal) where the pen was last activated.
- 2 Returns the y-coordinate (vertical) where the pen was last activated.
- 3 Returns a -1 if the pen button is being pressed. Returns a 0 if it is up.
- 4 Returns the last known valid x-coordinate (horizontal).
- 5 Returns the last known valid y-coordinate (vertical).
- 6 Returns the character row position where the pen was last activated.
- 7 Returns the character column position where the pen was last activated.
- 8 Returns the last known character row position.
- 9 Returns the last known character column position.

You must execute a PEN ON statement before executing the PEN function. If you do not, an Illegal function call error occurs.

Example

```
A = PEN(1)
```

returns the x-coordinate of the pen.

PEN/Trap

Statement

PEN *action*

Turns on, turns off, or temporarily halts light pen event trapping.

Action may be any of the following:

ON	enables event trapping
OFF	disables event trapping.
STOP	temporarily suspends event trapping.

Use the PEN/Trap statement in a light pen trap routine with the ON PEN statement to detect when the light pen has been activated.

The PEN ON statement turns on the trap. BASIC checks after each program line to see if the light pen has been activated. If so, BASIC transfers program control to the line number specified in the ON PEN GOSUB statement.

The PEN STOP statement temporarily halts light pen trapping. If the light pen is activated, BASIC does not transfer program control to the ON PEN GOSUB statement until you turn on trapping again by executing a PEN ON statement. BASIC remembers that the light pen was activated and branches to the subroutine immediately after trapping is turned on again.

The PEN OFF statement turns off light pen trapping. BASIC does not remember if the light pen was activated when trapping is turned on again.

You must also use PEN ON before executing the PEN function.

See ON PEN GOSUB for more information about light pen trapping.

PLAY

Statement

PLAY *string*

Plays the musical notes specified by *string*.

String is a string expression consisting of 1 or more single-character music commands. *String* must be enclosed in quotation marks.

The single character music commands are:

A - G plays notes A through G of one musical scale. You may include an optional number sign (#) or plus sign (+) to indicate a sharp note or a minus sign (-) to indicate a flat note. You may only specify sharp or flat notes that correspond to the black keys on a piano. The letters A, C, D, F, and G may be followed by a plus because they are followed by black keys on a piano. The letters A, B, D, E, and G may be followed by minus because they are preceded by black keys on a piano.

Ln sets the duration of the notes that follow. *n* may be a value in the range 1 to 64. Here are a few of the more common lengths:

- 1 indicates a whole note.
- 2 indicates a half note.
- 4 indicates a quarter note.
- 8 indicates an eighth note.
- 16 indicates a sixteenth note.

If you want to change the duration for only 1 note, place *n* immediately after the note, omitting the L. For example, A16 is equivalent to L16A.

On sets the current octave. There are 7 octaves, 0 through 6. Each octave starts with C and ends with B. Octave 3 starts with middle C. If you omit *n*, BASIC assumes Octave 4.

- > Changes the current octave to the next higher octave.
- < Changes the current octave to the next lower octave.
- Nn** plays a note. *n* may be in the range 0 to 84. In the 7 possible octaves, there are 84 notes. Instead of specifying the letter and the octave of the note, you may specify its number 1 to 84. Specifying zero means rest.
- Pn** rests. *n* may be in the range 1 to 64 and has the same meaning as *n* with the L option.
- Tn** sets the number of quarter notes in 1 minute. *n* may be in the range of 32 to 255. If you omit *n*, BASIC assumes 120 quarter notes in 1 minute. That is a moderate tempo. See the SOUND statement for information on beats per minute for common tempos.
- plays as a dotted note. BASIC plays the note one-half its length longer. You may use more than one dot after each note. BASIC scales the length of time accordingly. Dots may also appear after the P option to scale the length of the rest.
- MF** plays the music in the foreground, which includes sounds made by both PLAY and SOUND. This means that each subsequent note or sound does not start until the previous note or sound is finished. If you omit MF and MB, BASIC assumes MF.
- MB** plays the music in the background, which includes sounds made by both PLAY and SOUND. This means that each note or sound is placed in a buffer allowing the BASIC program to continue execution while music plays in the background. A maximum of 32 notes and/or rests can play in background at a time.

MN	sets “music normal”; each note plays 7/8 of the duration as set by the L option. If you omit MN and MS, BASIC assumes MN.
ML	sets “music legato”; each note plays the full duration as set by the L option.
MS	sets “music staccato”; each note plays 3/4 of the duration as set by the L option.
X <i>variable</i>;	executes a substring. The X command lets you execute a second substring from a string, much like GOSUB. You can have one string execute another, which executes a third, and so on. <i>Variable</i> is a string variable in your program that contains the substring you want to execute. <i>Variable</i> may contain an X command to execute another substring. The semicolon after the variable name is required.

With the O, N, P, and T commands, *n* may also be a numeric variable in your BASIC program. Do not space between the command and the *n* or between the command and the *variable*. You must include a semicolon after the variable name.

Example

```
10 PLAY "C4F.C8F8.C16F8.G16A2F2"
20 INPUT "CAN YOU NAME THAT TUNE ";A$
40 IF A$ = "THE EYES OF TEXAS" THEN GOTO 50 ELSE
   PRINT "TRY AGAIN": GOTO 10
50 PRINT "THAT'S RIGHT!"
```


PLAY

Function

PLAY(*number*)

Returns the number of notes currently in the background music queue.

The maximum number that can be returned is 32 because the buffer can hold a maximum of 32 notes and/or rests.

Number is a dummy argument.

The PLAY function returns a 0 when the program is running in music foreground mode.

See also SOUND.

Sample Program

```
10 PLAY "MB ABCDEFG"  
20 IF PLAY (0) = 4 GOTO 40  
30 GOTO 20  
40 PLAY "GFEDCBA"
```

Line 10 sends notes to the music buffer. When only 4 notes are left, Line 40 sends more notes to the buffer.

PLAY/Trap Statement

PLAY *action*

Turns on, turns off, or temporarily halts background music event trapping.

Action may be any of the following:

ON enables play event trapping.

OFF disables play event trapping.

STOP temporarily suspends play event trapping.

Use the PLAY/Trap statement in a background music trap routine with the ON PLAY GOSUB statement to detect when the background music queue goes from *number* to *number* minus 1.

The PLAY ON statement turns on the trap. BASIC checks the number of notes in the background music queue after each program line. If the number is equal to that in the ON PLAY() GOSUB statement, BASIC transfers program control to the line number specified.

The PLAY STOP statement temporarily halts background music trapping. If the number of notes equals the specified number, BASIC does not transfer program control to the ON PLAY() GOSUB statement until you turn on trapping again by executing a PLAY ON statement. BASIC remembers that the number of notes was equal and branches to the subroutine immediately after trapping is turned on again.

The PLAY OFF statement turns off background music trapping. BASIC does not remember if the number of notes in the queue is equal to the number specified when trapping is turned on again.

See ON PLAY() GOSUB for more information about background music trapping.

PMAP

Function

PMAP(*coordinate*,*action*)

Returns the physical or world coordinate for the specified coordinate.

Coordinate is any x- or y-coordinate. If *coordinate* is a physical coordinate, it must be within the limits of the screen. If *coordinate* is a world coordinate, it may be any single precision floating point number.

Action is one of the following:

- 0 returns the physical x-coordinate for the specified world coordinate.
- 1 returns the physical y-coordinate for the specified world coordinate.
- 2 returns the world x-coordinate for the specified physical coordinate.
- 3 returns the world y-coordinate for the specified physical coordinate.

Example

```
A = PMAP(200,0)
```

returns the physical x-coordinate of the world coordinate 200 and places it in A.

POINT/Graphics

Function

POINT (*x,y*)

POINT (*action*)

Returns the color number of a point on the screen or returns the current physical or world coordinates.

(*x,y*) specify the coordinates of the point. *x* is the horizontal point, and *y* is the vertical point. The *x* and *y* coordinates must be absolute values. If you specify a point that is out of range, BASIC returns a -1.

See Chapter 8, "Displaying Text and Graphics," for information on coordinates for the current screen modes.

Action is one of the following:

- 0 returns the current physical x-coordinate (horizontal).
- 1 returns the current physical y-coordinate (vertical).
- 2 If WINDOW is active, returns the world x-coordinate. Otherwise, returns the physical x-coordinate.
- 3 If WINDOW is active, returns the world y-coordinate. Otherwise, returns the physical y-coordinate.

When retrieving the color number, POINT returns the color number as it is defined in the current palette.

Example

```
10 SCREEN 1
20 IF POINT (1,1)<>0 THEN PRESET (1,1) ELSE PSET
   (1,1)
```

If point 1,1 is any foreground color, PRESET changes it to the background color. If the point is the background color, PSET changes it to Color 3.

```
10 SCREEN 1
20 X=POINT (0):Y=POINT(1)
30 PRINT X,Y
```

BASIC prints the coordinates of the graphics cursor.

POKE

Statement

POKE *memory location, data byte*

Writes *data byte* into *memory location*.

Both *memory location* and *data byte* must be integers. *Memory location* must be in the range -32768 to 65535.

POKE is the complementary statement of PEEK. The argument to PEEK is a memory location from which a byte is to be read.

PEEK and POKE are useful for storing data efficiently, loading assembly-language subroutines, and passing arguments (or results) to and from assembly-language subroutines.

See also VARPTR.

Example

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

writes a hexadecimal FF into memory location 5A00.

POS

Function

POS(*number*)

Returns the current column position of the cursor.

Number is a dummy argument.

POS returns a number in the range 1 to 80, indicating the current cursor-column position on the display.

Example

```
PRINT TAB(40) POS(0)
```

prints 40. The PRINT TAB statement moves the cursor to Position 40; therefore, POS(0) returns the value 40. (However, because a blank is inserted before the "4" to accommodate the sign, the "4" is actually at Position 41.)

Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) > 70 THEN IF A$ = CHR$(32) THEN A$
   = CHR$(13)
200 LPRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that you cannot correct mistakes). Your computer keyboard is the typewriter keyboard. Everything you type is printed on your printer. The program also makes sure that no word is divided between two lines.

PRINT

Statement

PRINT *data*[,*data*,...]

Prints numeric or string *data* on the display. You can substitute a question mark (?) in place of the word PRINT.

Data is any numeric or string constant or variable. If you omit *data*, BASIC prints a blank line. If you specify more than 1 data item in the statement, separate them by commas, semicolons, or spaces.

If you use commas, the cursor automatically advances to the next tab position before printing the next item. (BASIC divides each line into print zones containing 14 positions each, at columns 14, 28, 42, 56, and 70.)

If you use semicolons or spaces to separate the data items, PRINT prints the items without any spaces between them. BASIC begins the next PRINT item where the last one stopped.

If no trailing punctuation is at the end of the PRINT statement, the cursor drops to the beginning of the next line.

If BASIC tries to print a string longer than it can fit on the current line, it moves to the next line and prints the string.

Single precision numbers with 7 or fewer digits that can be accurately represented are printed in regular format rather than exponential format. For example, 1E-7 is printed as .0000001; 1E-8 is printed as 1E-08.

Double precision numbers with 16 or fewer digits that can be accurately represented are printed in regular format rather than exponential format. For example, 1D-15 is printed as .0000000000000001; 1D-16 is printed as 1D-16.

BASIC prints all numbers with a trailing blank and prints positive numbers with a leading blank. Negative numbers are preceded by a minus sign.

String constants must be enclosed in quotation marks.

Examples

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES"; "BETWEEN";  
      "THESE"; "WORDS"
```

displays DONOTLEAVESPACESEBETWEENTHESEWORDS

Sample Program

```
60 INPUT "ENTER THIS YEAR"; Y  
70 INPUT "ENTER YOUR AGE"; A  
80 INPUT "ENTER A YEAR IN THE FUTURE"; F  
90 N = A + (F - Y)  
100 PRINT "IN THE YEAR" F "YOU WILL BE" N "YEARS  
OLD"
```

Because F and N are positive numbers, PRINT inserts a space before and after them; therefore, your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in Line 100 by a comma:

```
100 PRINT "IN THE YEAR", F, "YOU WILL BE", N, "YEARS  
OLD"
```

BASIC would move to the next tab position after printing each data item.

PRINT USING

Statement

PRINT USING *format; data[,data,...]*

Prints data using a format you specified. This statement is especially useful for printing report headings, accounting reports, checks, or any other documents that require a specific format.

Format consists of 1 or more field specifier(s), or any alphanumeric character. *Format* must be enclosed in quotation marks.

Data may be string and/or numeric value(s). If you specify more than 1 data item in the statement, use the same separators as described in PRINT.

With PRINT USING, you may use certain characters called field specifiers, to format the field. You may use more than 1 field specifier, except as indicated.

Specifiers for String Fields:

! prints the first character in the string only.

```
PRINT USING "!"; "PERSONNEL"
```

BASIC prints P.

\spaces prints 2 + *n* characters from the string (*n* is the number of spaces between the slashes). If you type the backslashes without any spaces, BASIC prints 2 characters; with one space, BASIC prints 3 characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right.

```
PRINT USING "\bbb\"; "PERSONNEL"
```

BASIC prints PERSO.

& prints the string without modifications.

```
10 A$="TAKE":B$="RACE"  
20 PRINT USING "!";A$;  
30 PRINT USING "&";B$
```

When this program is run, BASIC prints TRACE.

Specifiers for Numeric Fields:

prints the same number of digit positions as number signs (#). Numbers are rounded as necessary.

You may insert a decimal point at any position. BASIC always prints the digits preceding the decimal point. If there is no number, BASIC prints a zero.

If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number.

```
PRINT USING "###.###";111.22
PRINT USING "###.###";.75
PRINT USING "####.###";876.567
```

BASIC prints %111.22, 0.75 and 876.57, respectively.

If the number of digits specified exceeds 24, an illegal function call occurs.

+ prints the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+###.## "; -98.45,3.50,22.22,-.9
```

BASIC prints: -98.45 +3.50 +22.22 -0.90

```
PRINT USING "###.##+ "; -98.45,3.50,22.22,-.9
```

BASIC prints: 98.45- 3.50+ 22.22+ -0.90-

(Note the use of spaces at the end of a format string to separate printed values.)

— prints a negative sign *after* negative numbers (and a space after positive numbers).

```
PRINT USING "###.##-"; -768.660
```

BASIC prints 768.7-

****** fills leading spaces with asterisks. The 2 asterisks also establish 2 more positions in the field.

```
PRINT USING "*****"; 44.0
```

BASIC prints ******44**

\$\$ prints a dollar sign immediately before the number. This specifies 2 more digit positions, one of which is the dollar sign. You may not use exponent format with \$\$.

```
PRINT USING "$$##.##"; 112.7890
```

BASIC prints **\$112.79**

****\$** fills leading spaces with asterisks and prints a dollar sign immediately before the number.

```
PRINT USING "**$##.##"; 8.333
```

BASIC prints *****\$8.33**

, prints a comma before every third digit to the left of the decimal point. The comma establishes another digit position.

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

BASIC prints **1,234.50**

^^^ prints in exponential format. The 4 exponent signs are placed after the digit position characters. You may specify any decimal point position. You may not use \$\$ with exponent format.

```
PRINT USING ".####^^^^"; 888888
```

BASIC prints **.8889E+06**

_ Prints next character as a literal character.

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

BASIC prints **!12.34!**

Sample Program

```
420 CLS: A$ = "***$##,#####.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P
470 CLS : PRINT "PAY TO THE ORDER OF ";
480 PRINT USING "!! !! "; F$; ". "; M$; ". ";
490 PRINT L$
500 PRINT :PRINT USING A$; P
```

In Line 480, each ! picks up the first character of one of the following strings (F\$, ".", M\$, and "." again). Notice the 2 spaces in "!!b!!b". These 2 spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A\$ for format and P for item list in Line 500. Any serious use of the PRINT USING statement would probably require the use of variables rather than constants, at least for data items. (We have used constants in our examples for the sake of better illustration.)

When the program above is run, the display shows:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,345.60 DOLLARS
```

PRINT#

Statement

PRINT# *buffer*,[**USING** *format*] *data*[,*data*,...]

Writes data items to a sequential disk file.

Buffer is the number assigned to the file when you opened it.

When you first open a file for sequential output, BASIC sets a pointer to the beginning of the file—that is where PRINT# starts writing the data items. At the end of each PRINT# operation, the pointer advances so that data items are written in sequence.

A `PRINT#` statement creates a disk image similar to the image a `PRINT` to the display creates on the screen. For this reason, be sure to delimit the data so that it will be input correctly from the disk.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

When you include the USING option, data is written to the disk in the format you specify. See PRINT USING.

Examples

```
If A = 123.45
PRINT# 1,A
```

writes this 9-byte character sequence to the file as:

0123.45 carriage return

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example:

```
A = 2300
B = 1.303
PRINT# 1, A,B
```

writes the data on disk as:

0 2300 0000000000 1.3030 carriage return

The comma tells BASIC to tab between A and B, which creates 10 extra spaces in the file. Generally you do not want to use up storage space this way, so you use semicolons instead of commas.

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

This time BASIC writes the data as:

```
123.45,1.303
```

An INPUT# statement reads this as 2 separate fields.

If string variables contain commas, semicolons, or leading blanks, enclose them in quotation marks. For example:

```
A$ = CAMERA, AUTOMATIC
B$ = 102382
PRINT# 1, A$; B$
```

writes the data as:

```
CAMERA                AUTOMATIC102382
```

An INPUT# statement reads this as 2 separate fields

```
A$ = CAMERA
B$ = AUTOMATIC102382
```

To separate these 2 strings properly in the file, write quotation marks using the hexadecimal representation CHR\$(34). For example:

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

BASIC writes the following image to the file:

```
"CAMERA,AUTOMATIC"102382"
```

The statement INPUT# 1, A\$, B\$ reads "CAMERA, AUTOMATIC" into A\$ and "102382" into B\$.

```
PRINT# 1, USING"!..!\BB\";A$;B$;C$
```

L.V.BEET

247

PSET/PRESET/Graphics Statement

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

Draws a point on the display.

The STEP option tells BASIC that the (*x,y*) coordinates are relative to the last point referenced.

(*x,y*) specify the coordinates in which to draw the point. *X* is the horizontal coordinate and *y* is the vertical coordinate.

Color specifies the color of the point.

See Chapter 8, "Displaying Text and Graphics" for information on coordinates and colors.

The only difference between the PSET and PRESET statements is the default values for *color*. If you use PSET, *color* defaults to the foreground color. If you use PRESET, *color* defaults to the background color.

Note: BASIC does not print and does not issue an error message for points the coordinate values of which are beyond the edge of the screen. However, values outside the integer range (-32768 to 32767) cause an overflow error.

Sample Program

```
5 SCREEN 1
10 FOR I=0 TO 100
20 PSET (I,I)
30 NEXT I 'draw a diagonal line to (100,100)
40 FOR I = 100 TO 0 STEP -1
50 PRESET (I,I),0
60 NEXT I
70 'clear the line by setting each pixel to 0
80 SCREEN 0
```

Lines 10 to 30 draw a diagonal line on the screen from the home position to Position 100,100. Lines 40 to 60 erase the line by drawing another line at the same position in the background color.

PUT

Statement

PUT [#]*buffer*[,*record*]

Puts a *record* in a direct access disk file.

Buffer is the number assigned to the file when you opened it. The number sign (#) is optional. It is provided for compatibility with other BASICs.

Record is the record number you want to write to the file. It is an integer in the range 1 to 16,777,215. If you omit *record*, the current record number is used.

If *record* is higher than the end-of-file record number, then *record* becomes the new end-of-file record number.

The first time you use PUT after opening a file you must specify the *record*. The first time you access a file via a particular buffer the next record is set equal to one greater than the last record accessed.

See Chapter 7, "Disk Files," for programming information.

Examples

PUT 1

writes the next record from Buffer 1 to a direct access file.

PUT 1,25

writes Record 25 from Buffer 1 to a direct access file.

PUT/Communications

Statement

PUT [#]*buffer,number*

Transfers data from the communications buffer to the communications line.

Buffer is the number assigned to the file when you opened it. The number sign (#) is optional. It is provided for compatibility with other BASICs.

Number is the number of bytes to transfer. It cannot exceed the value you used in the LEN option in the OPEN COM statement.

Note: Because of the low performance associated with telephone line communications, we recommend that you **not** use GET and PUT statements in such applications.

Example

```
PUT 2,80
```

transfers 80 bytes from communications buffer (Buffer 2) to the communications line.

Sample Program

```
10 OPEN "COM1:" AS 1
20 FIELD 1, 8 AS A$
30 OPEN "R",3,"data.fil",8
40 FIELD 3, 8 AS N$
50 FOR I = 1 TO 7
60 GET 3,I
70 LSET A$=N$
80 PUT 1,8
90 NEXT I
100 CLOSE
```

This program moves the data from the Data.fil file buffer to the communications buffer. Line 80 sends the data in the communications buffer to the communications line.

PUT/Graphics

Statement

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

Transfers an image stored in an array onto the screen.

You get the GET/Graphics and PUT/Graphics statements together for animation and high-speed object motion in the graphics modes. The GET/Graphics statement transfers the screen image described by specified points of the rectangle in the array. The PUT/Graphics statement transfers the image from the array to the display.

(*x,y*) specify the coordinates where the image begins. *x* is the horizontal coordinate and *y* is the vertical coordinate.

The *x* and *y* coordinates are the coordinates of the upper left corner of the image. If you omit *x* and *y*, BASIC begins the image at the last point referenced on the screen. See Chapter 8, "Displaying Text and Graphics," for information on coordinates for the current screen mode.

BASIC returns an `Illegal function call` error if the image is too large to fit on the current viewport.

Array is the array variable name that holds the image.

Action sets the type of interaction between the transferred image and the image already on the screen. *Action* may be PSET, PRESET, AND, OR, or XOR. If you omit *action*, BASIC assumes XOR. The following describes each type of action:

PSET transfers the data to the screen exactly as it was stored in the array.

PRESET produces an opposite image on the screen. In Screen Mode 2, white becomes black on the screen (and black becomes white). In Screen Mode 1 (the 4-color mode), the color value becomes the numeric opposite on the screen. This table shows the color displayed for each possible value:

Screen Mode 1

Array Value	Screen Color
0	3
1	2
2	1
3	0

AND transfers the image over existing image. The result is a logical AND of the array and the image on the screen.

OR superimposes an image onto an existing image. The result is a logical OR of the array and the image on the screen.

XOR inverts the points on the screen where a point exists in the array image. When an image is **PUT** against a complex background twice, the background is restored unchanged. This lets you move an object around the screen without obliterating the background.

Animation

To perform object animation, follow these steps:

1. Put the object on the screen using **XOR**.
2. Calculate the next position of the object.
3. Put the object on the screen a second time at the previous location to remove the previous image.
4. Repeat Step 1, putting the object at the next location.

If you do movement this way, the background is not changed. You can reduce flicker by minimizing the time between Steps 4 and 1 and by ensuring enough time delay between Steps 1 and 3. If you are animating more than 1 object, process every object at the same time, 1 step at a time.

If preserving the background is not important, you can perform animation using the PSET action rather than the XOR action. Leave a border around the image as large or larger than the maximum distance the object moves. When you move an object, this border effectively erases any points. This method may be faster than the XOR method described before, because only one PUT is required to move an object.

Sample Program

See the GET/Graphics statement.

RANDOMIZE

Function

RANDOMIZE [*number*]

Reseeds the random number generator.

Number may be an integer, or single- or double precision number. If you omit *number*, BASIC suspends program execution and prompts you for a number before executing RANDOMIZE:

Random Number Seed (-32768 to 32767)?

If the random number generator is not reseeded, the RND function returns the same sequence of numbers each time it is executed. To change the sequence of random numbers every time the RND function is executed, place a RANDOMIZE statement before the RND function.

You can use the TIMER function to ensure that the random number generator is reseeded with a different value each time BASIC executes the RANDOMIZE function. For example, the statement:

```
RANDOMIZE TIMER
```

uses the value returned by TIMER as the seed. TIMER returns the number of seconds that have elapsed since midnight or the last system reset. Because the seconds are constantly changing, *number* has a different value each time BASIC executes this statement.

Sample Program

```
10 CLS
20 RANDOMIZE TIMER
30 INPUT "PICK A NUMBER BETWEEN 1 AND 100"; A
40 B = INT(RND*100)
50 IF A=B THEN 80
60 PRINT "You lose, the answer is"B;"--try
again."
70 GOTO 20
80 PRINT "You picked the right number -- you
win."
```

READ

Statement

READ *variable[,variable,...]*

Reads values from a DATA statement and assigns them to *variables*.

BASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is assigned to the first variable; the second time, the second value is assigned to the second variable, and so on.

A single READ may access 1 or more DATA statements, or several READs may access the same DATA statement. If a program contains multiple DATA statements, BASIC reads them in the order they appear.

The values read must agree with the variable types specified in a list of variables; otherwise, a Syntax error occurs.

If the number of variables in the READ statement exceeds the number of elements in the DATA statement(s), BASIC returns an Out of DATA error message. If the number of variables specified is less than the number of elements in the DATA statement(s), the next READ statements begin reading data at the first unread element.

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

Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable T.

Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40 PRINT "NAME", "AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$, AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERS, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, "END"
```


REM

Statement

REM

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line, which lets you insert remarks in your program for documentation. Thus, when you look at a listing of your program, you can quickly interpret it.

If REM is used in a multistatement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for REM.

Sample Program

```
110 DIM V(20)
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
150 NEXT I
```

OR

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

RENUM

Statement

RENUM [*new line*][,*line*][,*increment*]

Renumbers the program currently in memory. You can renumber the entire program or renumber from a specific line to the end.

Line is the line in the program where BASIC starts renumbering. If you omit *line*, it renumbers the entire program.

New line is the new line number assigned to *line*. If you omit *new line*, BASIC starts numbering at Line 10.

Increment tells BASIC how to number the successive lines. If you omit *increment*, it increments by 10.

RENUM also changes all line number references appearing after GOTO, GOSUB, THEN, ON/GOTO, ON/GOSUB, ON ERROR GOTO, RESUME, and ERL.

You cannot use RENUM to change the order of program lines. For example, if a program has lines numbered 10, 20, and 30, the command RENUM 15,30 is illegal, since this would place Line 30 before Line 20.

Also RENUM cannot create line numbers greater than 65529. If you attempt to do this, BASIC returns an `Illegal function call` error and leaves the program unchanged.

If BASIC finds an undefined line number within the program, it prints a warning message, `Undefined line xxxx in yyyy`, where *xxxx* is the undefined line number and *yyyy* is the line where it appears. RENUM renumbers the program despite this warning message. It does not change the incorrect line number reference, but it does renumber *yyyy*.

Examples

RENUM

renumbers the entire program, using an increment of 10. The new number of the first line is 10.

RENUM 600, 5000, 100

renumbers from Line 5000 to the end of the program. The first renumbered line becomes 600, and an increment of 100 is used between subsequent lines.

RENUM 100, ,100

renumbers the entire program, starting with a new line number 100, and incrementing by 100s. Notice that the commas must be retained even though the middle argument is not used.

RESET

Statement

RESET

Closes all open files on all drives.

If a disk contains any open files, RESET writes all blocks in memory to disk.

RESET ensures that all files on all diskettes are closed before you remove them from the drives. RESET is the same as a CLOSE statement for each open file.

RESTORE

Statement

RESTORE [*line*]

Restores a program's access to previously read DATA statements.

Line is a line number that contains a DATA statement. If you specify *line*, the next READ statement accesses the first item in the specified DATA statement. If you omit *line*, BASIC resets to the first DATA statement in the program.

This lets your program reuse the same DATA lines.

Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA FIRST ITEM, SECOND ITEM
```

When you run this program, BASIC prints:

```
FIRST ITEM      FIRST ITEM
```

Because of the RESTORE statement in Line 170, the second READ statement starts over with the first DATA item.

RESUME

Statement

RESUME [*line*]
RESUME NEXT

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME 0 both cause the computer to return to the statement in which the error has occurred.

RESUME *line* causes the computer to branch to the specified line number.

RESUME NEXT causes the computer to branch to the statement following the point at which the error has occurred.

Examples

```
RESUME
```

If an error has occurred, this line transfers program control to the statement in which it has occurred.

```
RESUME 10
```

If an error has occurred, transfers control to Line 10.

Sample Program

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

RETURN

Statement

RETURN [*line*]

Returns control to the line immediately following the most recently executed GOSUB.

Line tells BASIC to return to a specific line in the program. If you omit *line*, BASIC goes to the line immediately following the GOSUB.

Use caution when specifying a line number with RETURN. Any other GOSUB, WHILE, or FOR statement remains active while a GOSUB subroutine is executing. If BASIC returns to a line number that is outside these loops, an error occurs because the loops were left incomplete.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

Example

```
RETURN 40
```

returns from the subroutine to Line 40 in the program.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A  
CIRCLE"  
340 INPUT "TYPE IN A VALUE FOR THE RADIUS"; R  
350 GOSUB 370  
360 PRINT "AREA IS" ; A: END  
370 A = 3.14 * R * R  
380 RETURN
```

RIGHT\$

Function

RIGHT\$(string,number)

Returns the specified number of characters from the far right portion of *string*.

Number is an integer in the range 1 to 255.

If *number* is equal to or greater than the length of *string*, BASIC returns the entire string.

Examples

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

Sample Program

```
850 RESTORE : ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2), : GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMON MANUFACTURING COMPANY, HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

RMDIR

Statement

RMDIR *pathname*

Removes (deletes) the directory specified by *pathname*.

Pathname is a standard directory specification as described in Chapter 1. If you omit the drive identifier, the directory is deleted from the current drive. If you omit the root directory symbol (\), the directory is deleted from the current directory.

The directory being deleted must be empty except for the "." and ".." symbols. Use the MS-DOS COPY command to move those files you want to save; then use KILL to remove all files from the directory.

Examples

```
RMDIR "A:\ACCTS\PAYABLE"
```

removes the directory PAYABLE from the ACCTS directory on Drive A.

```
RMDIR "\\ADDRESS"
```

removes the directory ADDRESS from the root directory on the current drive.

```
RMDIR "NAMES"
```

removes the directory NAMES from the current directory on the current drive.

RND

Function

RND[(*number*)]

Returns a random number in the range 0 and 1.

BASIC uses the current seed when generating a random number and produces the same sequence of random numbers each time the program is run unless you reseed the random number generator. Use the RANDOMIZE statement to reseed the random number generator.

If *number* is negative, RND starts the sequence of random numbers at the beginning. If *number* is 0, RND repeats the last number generated. If you omit *number* or specify a positive value, RND returns the next number in the sequence.

Example

```
PRINT RND(1)
```

prints the next decimal fraction in the sequence.

Sample Program

```
10 FOR I = 1 TO 5  
20 PRINT INT(RND*100);  
30 NEXT I
```

This program produces 5 random integers. Line 20 converts the decimal fraction returned by RND to a real number and truncates the real number to an integer.

RSET

Statement

RSET *field name* = *data*

Sets *data* in a direct access buffer *field name* in preparation for a PUT statement.

Field name is a string variable defined in a FIELD statement.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

RUN Statement

RUN [*line*]

RUN *pathname*[,R]

Executes a program.

Line is the program line where BASIC begins execution. If you omit *line*, BASIC executes the program from the beginning.

Pathname specifies the disk file for BASIC to load into memory and execute.

If you specify the R option, BASIC does not close the open files before loading the new program into memory. If you omit the R option, BASIC closes all open files before loading the program.

RUN automatically clears all variables.

Examples

```
RUN
```

starts execution at the beginning of the program.

```
RUN 100
```

starts execution at Line 100.

```
RUN "program.a"
```

loads and executes Program.a.

```
RUN "\WORD\editdata", R
```

loads and executes Editdata from the WORD directory without closing any open files.

SAVE**Statement****SAVE** *pathname* [,A]**SAVE** *pathname* [,P]

Saves a program on disk with the specified name.

Pathname is a standard file specification as described in Chapter 1. When you save a file to disk, you must specify the filename. If the file already exists on disk, its contents are lost when the file is re-created.

The A option tells BASIC to save the program in ASCII format. If you omit the A option, BASIC saves the file in a compressed format.

The compressed format takes less disk space than ASCII format. Also BASIC can save and load in compressed format faster than in ASCII format. BASIC programs are stored in RAM using compressed format.

Use the ASCII format if you plan to use the MERGE command to merge the program with another. Also, data programs that be read by other programs usually must be in ASCII.

When using the ASCII option, be sure your program has no embedded line feeds; otherwise, the computer will not be able to read it properly. Embedded line feeds are produced by typing **CTRL****J** in a program line.

For compressed-format programs, a useful convention is the extension .bas. For ASCII-format programs, use .txt.

The P option protects the file by saving it in an encoded binary format. When a protected file is later run (or loaded), any attempt to list or edit it fails. The only operations that you can perform on a protected file are RUN, LOAD, and CHAIN.

Examples

```
SAVE "A:file1.bas"
```

saves the resident program in compressed format as File1.bas. The file is placed on Drive A: in the current directory.

```
SAVE "\EDUC\mathpak.txt", A
```

saves the resident program in ASCII form, using the name Mathpak.txt, on the current drive in the directory EDUC.

SCREEN

Function

SCREEN (*row*, *column*, [*number*])

Returns the ASCII code for the character at the specified row and column.

Row is an integer in the range 1 to 25.

Column is an integer in the range 1 to 80. *Column* must be in the range for the current screen mode.

Number is applicable only for text mode. If you specify a non-zero *number*, SCREEN returns the color attribute, rather than the ASCII code, of the character. The attribute is in the range of 0 to 255, and can be translated as follows:

attribute MOD 16 = foreground color (((*attribute* - foreground)/16)MOD 128) = background color

(*number*>127) is true (-1) if the character is blinking and false (0) if it is not.

In the graphics modes, if the location does not contain a standard ASCII character, BASIC returns a value of zero.

Sample Program

```
10 LOCATE 20,20
20 PRINT "ROBBIE"
30 A = SCREEN(20,20):B = SCREEN(20,21)
40 PRINT A,B
```

Line 10 positions the cursor to Row 20, Column 20. Line 20 prints the message at the current cursor position. Line 30 stores the ASCII code for "R" in the variable A and the ASCII code for "O" in variable B. Line 40 prints:

82

79

SCREEN**Statement**

SCREEN [*mode*][,*burst*][,*active page*][,*display page*]]

Sets the screen attributes to be used by all other graphics statements (CIRCLE, LINE, DRAW, POINT, PSET, PRESET).

Mode is an integer in the range 0 to 2 that sets the valid coordinates and the number of colors you can use. Screen mode descriptions are given in Chapter 8, "Displaying Text and Graphics."

When you change from one mode to another, BASIC stores the new screen mode, erases the video display, and sets the foreground color to white and the background and border colors to black.

Burst activates or de-activates color in Screen Mode 0 or 1. Set *burst* to one of these values:

Mode	Activate	De-Activate
0	1	0
1	0	1

Burst has no effect in Screen Mode 2, the black-and-white graphics mode.

Active page is an integer that selects the video page to which BASIC will write. All output statements to the screen go to the selected *active page*. The range depends on the screen mode and the amount of video memory available. If you omit *active page*, BASIC assumes the current active page. *Active page* is initially set to Page 0.

Display page is an integer that selects the video page for BASIC to display. The range is the same as *active page*. If you omit *display page*, BASIC uses the same page as *active page*. BASIC automatically sets *display page* to *active page* if the program halts because of an END or STOP statement or because of an error.

All video pages share one cursor. Therefore, when switching active pages, you should save the cursor position with the POS

and CSRLIN statements. Then when you return to an active page, you can restore the cursor with the LOCATE statement.

If you omit any parameter (except *display page*), BASIC continues to use the current value for that particular parameter.

For more information on the graphics statements and video pages, see Chapter 8, "Displaying Text and Graphics."

Examples

```
10 SCREEN 0,0
```

selects text mode with color off.

```
60 SCREEN 2
```

changes to high resolution, 2-color, graphics mode.

SGN

Function

SGN(*number*)

Determines *number*'s sign.

If *number* is a negative number, SGN returns -1.

If *number* is a positive number, SGN returns 1.

If *number* is zero, SGN returns 0.

Examples

```
Y = SGN(A * B)
```

determines the sign of the expression $A * B$, and passes the appropriate number (-1,0,1) to Y.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SHELL

Advanced Statement

SHELL [*command*]

Loads and executes another program (*.EXE* or *.COM*) or an internal command as a child process to the original program. After the child process ends, control returns to the BASIC program at the statement following the SHELL statement.

Command is a string expression containing the name of the program you want to run. You may also specify command arguments on the command line. Use a space to separate arguments from the program name. If you omit *command*, SHELL transfers control to COMMAND. You can now execute MS-DOS commands as allowed by COMMAND. To return to BASIC, use the MS-DOS EXIT command.

SHELL sends the *command* information to COMMAND.COM, the MS-DOS command processor. If you omit the extension in the program name, COMMAND looks for the program with a *.COM* extension, then with an *.EXE* extension and finally with a *.BAT* extension. If COMMAND still cannot find the program, it issues a File not found error to SHELL.

Note: Do not specify BASIC as the command string of SHELL. If you do, BASIC might not function properly.

For more information on child processes and COMMAND.COM, see the *MS-DOS Reference* and the *Programmer's Reference* manuals for your computer. They are available through your Radio Shack Computer Store.

Examples

```
SHELL
```

transfers control to COMMAND.COM. You can execute MS-DOS commands such as:

```
DIR  
TIME
```

and then type EXIT to return to BASIC.

The following command uses redirection of input and output and the MS-DOS SORT command.

```
SHELL"SORT <data.in >data.out"
```

sorts the text from Data.in and writes it to Data.out.

SIN

Function

SIN(*number*)

Returns the sine of *number*.

SIN returns the angle (in radians) whose cosine is *number*.

Number must be in radians. To obtain the sine of *number* when *number* is in degrees, use SIN(*number* • PI/180), where PI equals 3.141593.

BASIC always returns the result as a single precision number unless you set the /D switch when starting up BASIC.

Examples

```
PRINT SIN(7.96)
```

prints .9943854.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN(A * .01745329)
```

SOUND

Statement

SOUND *frequency, duration*

Generates a sound with the *frequency* and *duration* specified.

When a SOUND statement is producing sound, the program continues to execute. See the PLAY statement for more information about executing program lines during SOUND.

frequency specifies the desired tone in Hertz, and is an integer in the range 37 to 32767. The frequency 32767 is treated as the silence frequency. To create periods of silence, use SOUND 32767, *duration*. Possible frequencies are given later.

Duration is a numeric expression in the range 0.027 to 65535, specifying the duration in clock ticks. Clock ticks occur 18.2 times per second. If *duration* is 0, BASIC turns off any currently running SOUND statement. Otherwise, BASIC completes the first SOUND statement before executing the next one.

You can buffer sound by specifying MB in the PLAY statement. Doing this ensures that execution does not stop when BASIC encounters another SOUND statement.

This statement can be especially useful in educational applications. For example, you can have the computer respond with a sound if a user has answered a program's prompt incorrectly (or vice versa).

You can use the SOUND or PLAY statements to generate musical notes from your computer. The following chart shows the frequency you specify to generate the notes in the octave above middle C. Middle C is the first note in the chart.

Note	Frequency
------	-----------

C	523.25
D	587.33
E	659.26
F	698.46
G	783.99
A	880.00
B	987.77
C	1046.50

To generate notes in the octave below middle C, find the frequency of the note's letter in the chart and divide that number by 2. For example, the note A in the octave below middle C has a frequency of 440.00.

To generate notes in the octave above middle C, find the frequency of the note's letter in the chart and multiply that number by 2. For example, the note A in the octave above middle C has a frequency of 1760.00.

There are 1092 clock ticks per minute. To determine the number of clock ticks for 1 beat, divide the beats per minute into 1092. The chart below shows the number of clock ticks for some typical tempos.

Tempo	Beats per minute	Ticks per minutes
Largo	40- 60	27.3 -18.2
Larghetto	60- 66	18.2 -16.55
Adagio	66- 76	16.55-14.37
Andante	76-108	14.37-10.11
Moderato	108-120	10.11- 9.1
Allegro	120-168	9.1 - 6.5
Presto	168-208	6.5 - 5.25

Sample Program

```
10 INPUT "IN HONOR OF WHOM WAS THE CONTINENT OF  
AMERICA NAMED"; A$  
20 IF A$="AMERIGO VESPUCCI" THEN SOUND 500,50  
ELSE GOTO 40  
30 PRINT "THAT'S RIGHT!": END  
40 SOUND 37,2 : PRINT "THE CORRECT ANSWER IS  
AMERIGO VESPUCCI"
```

SPACE\$

Function

SPACE\$(*number*)

Returns a string of *number* spaces.

Number must be in the range 0 to 255. If *number* is greater than the width of the device, SPC uses *number* modulo *width*. (See Chapter 5 for an explanation of modulo arithmetic.)

Example

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9)
"QUANTITY"
```

prints DESCRIPTION, 4 spaces, TYPE, 9 spaces, QUANTITY.

Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```


SPC

Function

SPC(*number*)

Skips *number* spaces in a PRINT statement..

Number is in the range 0 to 255. A semicolon is assumed to follow the SPC(*number*) command.

You may use SPC only with PRINT, LPRINT, or PRINT# .

See also SPACE\$.

Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints:

```
HELLO
```

```
THERE
```

SQR

Function

SQR(*number*)

Returns the square root of *number*.

Number must be greater than zero.

BASIC always returns the result as a single precision number unless you specified the /D switch when starting up BASIC.

Example

```
PRINT SQR(155.7)
```

prints 12.47798.

Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

STICK**Function****STICK** (*action*)

Returns the coordinates of the joysticks.

Action may be one of the following:

- 0 reads all 4 coordinates, and returns the horizontal (x) coordinate for the left joystick.
- 1 returns the vertical (y) coordinate for the left joystick.
- 2 returns the horizontal (x) coordinate for the right joystick.
- 3 returns the vertical (y) coordinate for the right joystick.

The coordinates returned by STICK(1), STICK(2), and STICK(3) are those previously read by STICK(0).

Sample Program

The following program continually displays the coordinates of the right joystick.

```
10 CLS
20 LOCATE 1,1
25 T=STICK(0)
30 PRINT"B: ";STICK(2)
40 PRINT"B: ";STICK(3)
50 GOTO 20
```

STOP

Statement

STOP

Stops program execution.

When BASIC encounters a STOP statement, it prints the message `BREAK IN xxxx`, where `xxxx` is the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

Use the CONT statement if you want to resume execution. If the program itself has been altered during the break, you cannot use CONT.

Unlike the END statement, STOP does not close files.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number in the range 1 to 10 is assigned to X and then program execution halts at Line 2270. You can now examine the value X with `PRINT X`. Type CONT to start the cycle again.

STR\$

Function

STR\$(*number*)

Converts *number* to a string.

If *number* is positive, STR\$ places a blank before the string. If *number* is negative, STR\$ places a minus sign (-) before the string.

While arithmetic operations may be performed on *number*, only string functions and operations may be performed on the string.

The complementary function to STR\$ is VAL.

Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S\$.

Sample Program

```
10 A = 1.6 : B# = A : C# = VAL(STR$(A))
20 PRINT "REGULAR CONVERSION" TAB(40) "SPECIAL
   CONVERSION"
30 PRINT B# TAB(40) C#
```

STRIG

Statement

STRIG ON STRIG OFF

Enables the STRIG function.

STRIG ON

STRIG ON lets you execute STRIG function statements to return the status of the joystick buttons. If you attempt to execute a STRIG function before you execute a STRIG ON statement, BASIC issues an `Illegal function call` error.

STRIG OFF

If you execute a STRIG OFF statement, you cannot execute the STRIG function. Executing a STRIG function after a STRIG OFF statement results in an `Illegal function call` error.

When you load BASIC, the default is STRIG OFF and you cannot execute STRIG/Function statements.

You cannot place a STRIG function in a subroutine that you branch to as a result of an `ON STRIG() GOSUB` statement. BASIC does not keep track of which button was pressed after the `ON STRIG() GOSUB` statement is executed. If you wish to trap both buttons and perform a different procedure for each button, you must execute a STRIG/Trap for each button, and you must branch to different subroutines with different `ON STRIG() GOSUB` statements.

See the STRIG function, STRIG/Trap, and `ON STRIG() GOSUB` for additional information on joystick trapping.

STRIG

Function

STRIG(*number*)

Returns the status of joystick buttons.

Number is an integer in the range 0 to 7 to test the status of the joystick buttons.

Variable is a numeric variable to receive the value returned by *number*.

Each *number* tests for a different status of the buttons and returns a numeric value in *variable* regarding the results of the test. The *numbers* and their functions are:

- 0 Tests to see if Trigger 1 on the left joystick has been pressed and released since the last STRIG(0) function was executed. BASIC returns a -1 if it has been pressed and a 0 if not.
- 1 Tests to see if you are currently pressing Trigger 1 on the left joystick. BASIC returns a -1 if you are pressing it and a 0 if not.
- 2 Tests to see if Trigger 1 on the right joystick has been pressed and released since the last STRIG(2) function was executed. BASIC returns a -1 if it has been pressed and a 0 if not.
- 3 Tests to see if you are currently pressing Trigger 1 on the right joystick. BASIC returns a -1 if you are pressing it and a 0 if not.
- 4 Tests to see if Trigger 2 on the left joystick has been pressed and released since the last STRIG(4) function was executed. BASIC returns a -1 if it has been pressed and a 0 if not.
- 5 Tests to see if you are currently pressing Trigger 2 on the left joystick. BASIC returns a -1 if you are pressing it and a 0 if not.

- 6 Tests to see if Trigger 2 on the right joystick has been pressed and released since the last STRIG(6) function was executed. BASIC returns a -1 if it has been pressed and a 0 if not.
- 7 Tests to see if you are currently pressing Trigger 2 on the right joystick. BASIC returns a -1 if you are pressing it and a 0 if not.

You must execute a STRIG ON statement before you can execute a STRIG function. If you attempt to execute a STRIG function before you execute a STRIG ON, BASIC issues an illegal function call error. See STRIG/Trap.

You cannot place a STRIG function in a subroutine that you branch to as a result of an ON STRIG() GOSUB statement. BASIC does not keep track of which button was pressed after the ON STRIG() GOSUB statement is executed. If you wish to trap both buttons and perform a different procedure for each button, you must execute a STRIG/Trap for each button, and you must branch to different subroutines with different ON STRIG() GOSUB statements.

Sample Program

This program tells BASIC to beep whenever the trigger on the left joystick is pressed.

```
10 STRIG ON
20 IF STRIG(0) THEN BEEP
30 GOTO 20
```


STRIG/Trap**Statement**

STRIG(*number*) ON
STRIG(*number*) OFF
STRIG(*number*) STOP

Turns on, turns off, or temporarily halts joystick trapping.

Number is a value of 0, 2, 4, or 6 to indicate the joystick button you are trapping:

- 0 indicates Trigger 1 on the left joystick.
- 2 indicates Trigger 1 on the right joystick.
- 4 indicates Trigger 2 on the left joystick.
- 6 indicates Trigger 2 on the right joystick.

STRIG() ON

STRIG() ON enables joystick trapping with the ON **STRIG()** GOSUB statement. If you execute a **STRIG()** ON statement, BASIC checks after every program statement to see if you pressed a joystick button. If you press a joystick button, BASIC transfers program control to the line number specified in the ON **STRIG()** GOSUB statement. See ON **STRIG()** GOSUB.

Note: Do not confuse the **STRIG/Trap** statement with the **STRIG** function statement. These are separate statements that perform distinct functions in BASIC.

STRIG() STOP

STRIG() STOP temporarily halts joystick trapping. If you press a joystick button after a **STRIG()** STOP statement is executed, BASIC does not transfer program control to the subroutine until trapping is turned on again with a **STRIG()** ON statement. BASIC remembers that the joystick buttons were pressed and transfers program control to the subroutine immediately after joystick trapping is turned on again.

STRIG OFF

STRIG() OFF turns off joystick trapping with the ON STRIG() GOSUB statement.

When you load BASIC, STRIG() OFF is the default, because joystick trapping slows program execution. Therefore, if you execute a STRIG() ON statement to enable joystick button trapping, we recommend that you also execute a STRIG() OFF statement when you no longer need to check for joystick button activity.

If you press a joystick button after a STRIG() OFF statement is executed, BASIC does not remember that the joystick buttons were pressed when joystick trapping is turned on again.

Example

See STRIG.

STRING\$

Function

STRING\$(*number,character*)

Returns a string containing the specified number of *character*.

Number must be in the range 0 to 255.

Character is a string or an ASCII code. If you use a string constant, you must enclose it in quotation marks. All the characters in the string have either the ASCII code, or the first letter of the string specified.

STRING\$ is useful for creating graphs or tables.

Examples:

```
B$ = STRING$(25, "X")
```

puts a string of 25 "X"s into B\$.

```
PRINT STRING$(50, 10)
```

prints 50 blank lines on the display, because 10 is the ASCII code for a line feed.

Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN 3 NUMBERS BETWEEN 33 AND
      159"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20, N1):
      NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2): NEXT
      J
1080 PRINT STRING$(80, N3)
```

This program prints 3 strings. Each string has the character corresponding to one of the ASCII codes provided.

SWAP

Statement

SWAP *variable1,variable2*

Exchanges the values of 2 variables.

You may swap variables of any type (integer, single precision, double precision, or string). However, both must be of the same type; otherwise, a `Type mismatch error` results.

Either or both variables may be elements of arrays. If one or both of the variables are non-array variables that have not been assigned values, an `Illegal function call error` results.

Example

```
SWAP F1#, F2#
```

swaps the values of F1# and F2#. The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
10 A$="ONE ":B$="ALL ":C$="FOR "  
20 PRINT A$ C$ B$  
30 SWAP A$, B$  
40 PRINT A$ C$ B$
```

When run, the program displays:

```
ONE FOR ALL  
ALL FOR ONE
```

SYSTEM

Statement

SYSTEM

Returns you to the MS-DOS command level.

BASIC closes all files before returning to MS-DOS. Your resident BASIC program is not retained in memory.

Examples

SYSTEM

returns you to MS-DOS. Your resident BASIC program is lost.

TAB

Function

TAB(*number*)

Spaces to position *number* on the display.

Number must be in the range 1 to 255 and specifies the character position to which to tab. The leftmost position is 1, and the rightmost position is the set width minus 1 (WIDTH-1).

If the current print position is already beyond space *number*, TAB goes to that position on the next line.

You cannot use TAB to move the cursor to the left.

You cannot use TAB more than once in a print list.

You may use TAB only with the PRINT and LPRINT statements.

Sample Program

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

When you run this program, the display shows:

NAME	AMOUNT
G.T.JONES	\$25.00

TAN**Function****TAN(*number*)**

Returns the tangent of *number*.

Return the angle (in radians) whose arc tangent is *number*.

Number must be in radians. To obtain the tangent of *number* when it is in degrees, use TAN (*number* * PI/180), where PI equals 3.141593.

BASIC always returns the result as a single precision number unless you set the /D switch when starting up BASIC.

Example

```
PRINT TAN(7.96)
```

prints -9.396959.

Sample Program

This programs asks you to input an angle in degrees and returns the tangent in radians.

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

TIME\$

Function

TIME\$[= *string*]

Sets or retrieves the current time.

String is a literal, enclosed in quotation marks, that sets the time by assigning its value to TIME\$. If you omit *string*, BASIC retrieves the current time.

BASIC uses a 24-hour clock. For example, it sets 8:15 P.M. as 20:15:00.

Setting the Time

You set the time in the following format:

hh:mm:ss

The hours (*hh*) may be any number 0-23.

The minutes (*mm*) and the seconds (*ss*) may be any number 0 through 59.

If you omit the minutes, minutes **and** seconds default to zero. If you omit the seconds, seconds default to zero.

Although you may omit leading zeros in each of the values, you must include at least 1 digit of the preceding value. For example, you may type 1:5 to set the the time to 1:05 A.M. However, :5 is invalid.

Retrieving the Time

BASIC always returns the time in the 8-character (hh:mm:ss) format, with leading zeros. You may set the time before you enter BASIC. If you do not set the time at the MS-DOS time prompt or with the TIME\$ statement, BASIC returns the length of the time that has elapsed since you turned on the computer.

Examples

```
TIME$ ="14:15"
```

sets the current time to 14:15:00.

```
TIME$ = "3:3:3"
```

sets the current time to 03:03:03.

```
A$=TIME$
```

assigns the current time to the variable A\$.

```
PRINT TIME$
```

prints the current time.

TIMER

Function

TIMER

Returns the number of seconds since midnight or since the last system reset.

BASIC always returns a single precision number.

You can use **TIMER** as the argument for the **RANDOMIZE** statement to reseed the random number generator. See **RANDOMIZE** for more information.

Example

```
A = TIMER
```

stores the number returned by **TIMER** into variable **A**.

TIMER/Trap **Statement**

TIMER *action*

Turns on, turns off, or temporarily halts timer event trapping.

Action may be any of the following:

ON	enables timer event trapping.
OFF	disables timer event trapping.
STOP	temporarily suspends timer event trapping.

The **TIMER ON** statement turns on the trap. BASIC checks the the value of timer after each program line. If the number is equal to that in the **ON TIMER()** **GOSUB** statement, BASIC transfers program control to the line number specified.

The **TIMER STOP** statement temporarily halts timer trapping. If the timer equals the specified number, BASIC does not transfer program control to the **ON TIMER()** **GOSUB** statement until you turn on trapping again by executing a **TIMER ON** statement. BASIC remembers that the timer value was equal and branches to the subroutine immediately after trapping is turned on again.

The **TIMER OFF** statement turns off timer trapping. BASIC does not remember if the value of timer equals the number specified when trapping is turned on again.

Sample Program

See **ON TIMER()** **GOSUB** for an example.

TROFF, TRON

Statements

TROFF TRON

Turn the trace function on/off.

TRON turns on the tracer and TROFF turns it off.

The tracer lets you follow program flow. This is helpful for debugging and for analyzing the execution of a program. After a program is debugged, you can remove the TRON and TROFF statements.

Each time the program advances to a new line, the tracer displays that line number inside a pair of brackets.

Sample Program

```
2290 TRON
2300 X = X • 3.14159
2310 TROFF
```

Lines 2290 and 2310 assure you that Line 2300 is actually being executed, because [2300] is printed on the display each time it is executed.

```
5 TRON
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINT J;K;L
50 K=K+10
60 NEXT J
70 TROFF
80 END
```

When you run this program, BASIC prints:

```
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
```

USR

Function

USR*[number](argument)*

Calls a user's assembly-language subroutine identified with *number* and passes *argument* to that subroutine.

The *number* you specify must be the same as the corresponding DEF USR statement for that routine. If you omit *number*, BASIC assumes zero.

USR lets you call as many as 10 assembly-language subroutines and then continue execution of your BASIC program.

Before you can execute a USR function call, you must define the subroutine's address in a DEF SEG and DEF USR statement. The DEF SEG defines the address of the segment containing the subroutine. The DEF USR statement defines the subroutine being called and its offset from the beginning of the segment set by DEF SEG. See DEF SEG, DEF USR, and the section "Interfacing with Assembly-Language Subroutines" in Chapter 11.

VAL

Function

VAL(*string*)

Calculates the numerical value of *string*.

VAL is the inverse of the STR\$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

VAL terminates its evaluation on the first character that has no meaning in a numeric value.

If the string is nonnumeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 123400000.

Sample Programs

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

This program searches for zip codes within the specified ranges to determine if they are within Long Beach or "out of state."

VARPTR

Function

VARPTR (*variable*)

VARPTR ([#]*buffer*)

Returns the offset into BASIC's data segment of a variable or a disk buffer.

Variable is a numeric or string variable.

Buffer is the number assigned to the file when you opened it. The number sign is optional. It is provided for compatibility with other BASICs.

VARPTR can help you locate a value in memory. When used with *variable*, it returns the address of the first byte of data identified with *variable*. See the section "How Variables are Stored" in Chapter 11 for the format.

If the *variable* you specify has not been assigned a value, an `Illegal function call` occurs.

When used with sequential access files, VARPTR returns the starting address of the disk buffer. When used with direct access files, VARPTR returns the address of the FIELD buffer.

If you specify a *buffer* that was not allocated when loading BASIC, a `Bad file number error` occurs. (See Chapter 2 for information on how to load BASIC.)

The offset returned is an integer in the range -32768 to 32767. It is always an offset into BASIC's data segment, regardless of whether you have executed a `DEF SEG` to change the segment.

VARPTR is used primarily to pass a value to an assembly language subroutine via `USR`. Since VARPTR returns an offset that indicates where the value of a variable is stored, you can pass this address to an assembly-language subroutine as the argument of `USR`. The subroutine can then extract the contents of the variable with the help of the address that you have supplied to it.

If VARPTR returns a negative address, add it to 65536 to obtain the actual address.

VARPTR\$

Function

VARPTR\$(*variable*)

Returns a character form of the memory address of the variable. VARPTR\$ is primarily used with PLAY and DRAW in programs that are later executed.

Variable is a numeric or string variable.

VARPTR\$ returns a 3-byte string:

byte 0	=	<i>type</i>
byte 1	=	<i>low byte of address</i>
byte 2	=	<i>high byte of address</i>

Type is 2 for integer variables, 3 for string variables, 4 for single precision variables, and 8 for double precision variables.

Note: Because array and string addresses and file data blocks change whenever you assign a new variable, do not store the contents of VARPTR\$ into a variable.

Example

```
10 PLAY "X" +VARPTR$(A$)
```

uses the PLAY subcommand X, plus the contents of A\$, as the argument for PLAY.

VIEW/Graphics

Statement

VIEW [SCREEN] [(*x1*,*y1*)-(*x2*,*y2*)[, [*color*][, [*border*]]]

Creates a viewport that redefines the screen parameters. This defined area, a window, becomes the only place you can draw graphic displays.

(*x1*,*y1*) specifies the upper-left coordinates for the rectangular viewport.

(*x2*,*y2*) specifies the lower-right coordinates for the rectangular viewport.

All coordinates must be within the limitations of the screen.

Color lets you fill in the specified viewport with the specified color. See Chapter 8, "Displaying Text and Graphics," for information on color.

Border is an integer expression that specifies the color for the boundary line around the viewport (assuming there is enough space for the line). The range is the same as that for *color*. If you omit *border*, no boundary line is drawn.

SCREEN specifies that all coordinates used in drawing are absolute to Point 0,0 on the screen. If you omit SCREEN, all coordinates specified are relative to the viewport coordinates.

If you omit all options, BASIC sets the viewport to define the entire screen.

Examples

```
VIEW (10,10)-(100,100)
```

sets up a viewport with the upper-left corner at 10,10 and the lower-right corner at 100,100. Since SCREEN is omitted, all subsequent coordinates are relative to the viewport. For example, PSET (5,5),3 actually sets point 15,15.

```
VIEW SCREEN (20,25)-(100,150)
```

sets up a viewport. Because SCREEN is specified, all subsequent coordinates are absolute. For example, PSET (5,5),3 does not appear because it is outside the viewport. PSET (30,30),3 is within the viewport.

Notes:

- BASIC ignores any points that are outside the viewport's limits.
- RUN, SCREEN, and WINDOW statements, without parameters, define the entire screen as the viewport.
- CLS clears only the active viewport.

Sample Program

```
10 SCREEN 1
20 VIEW (10,10)-(200,100),2
30 PSET (100,50)
40 DRAW "L40 E20 F20"
```

Line 20 sets the viewport. Line 30 sets the starting point for the DRAW statement in Line 40.

VIEW PRINT

Statement

VIEW PRINT *top line* TO *bottom line*

Creates a text viewport that redefines the text screen parameters. All statements and functions that normally function within the text viewport now function in the new text screen parameters. Cursor movement and scrolling are also limited to the text viewport.

Top line specifies the first line of the text viewport. It may be in the range 1 to 25, but must be less than *bottom line*.

Bottom line specifies the last line of the text viewport. It may be in the range 1 to 25, but must be greater than *top line*.

If you omit all parameters, VIEW PRINT defines the entire screen as the text viewport.

Example

```
VIEW PRINT 1 TO 15
```

BASIC defines the first 15 lines of the video as the text viewport. All cursor movements, scrolling, and text screen functions and statements are limited to these boundaries.

WAIT

Statement

WAIT *port*, *number1* [,*number2*]

Suspends program execution until a machine input *port* develops a specified bit pattern. (A port is an input/output location.)

Number1 and *Number2* are integers in the range 0 to 255.

BASIC reads the data at the specified port and XORs it with *number2*, if given. If you omit *number2*, BASIC XORs the data with zero. BASIC then ANDs the result with *number1*. If the result is zero, BASIC starts again with reading the data at the port again. If the result is nonzero, BASIC continues with the next statement.

It is possible to enter an infinite loop with the WAIT statement. In this case, you must restart the machine manually. To avoid this, WAIT must have the specified value at port number during some point in program execution.

Example

```
100 WAIT 32,2
```

WHILE...WEND**Statement****WHILE** *expression*
WEND

Executes a series of statements in a loop as long as a given condition is true.

Expression is any numeric or string expression, usually making logical or relational comparisons.

If *expression* is true, BASIC executes the statements after the WHILE statement until it encounters a WEND statement. Then BASIC returns to the WHILE statement and checks *expression*. If it is still true, BASIC repeats the process. If it is not true, execution resumes with the statement following the WEND statement.

You may nest WHILE/WEND loops to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a WHILE without WEND error, and an unmatched WEND causes a WEND without WHILE error.

Sample Program

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

This program sorts the elements in array A\$. Control falls out of the WHILE loop when no more swaps are performed on Line 130.

WIDTH

• Statement

WIDTH [LPRINT] *size*

WIDTH *buffer, size*

WIDTH *device, size*

Sets the line width in number of characters for the display, line printer, or communication channel.

Size may be an integer in the range 0 to 255 that specifies the number of characters in a line. For the screen, *size* may be only 40 or 80.

Buffer is an integer in the range 0 to 15 and specifies the *buffer* used in the OPEN statment.

When you specify *buffer*, BASIC changes the width immediately. This lets you change the width when the file is open. To return to the previous width, execute another WIDTH statement.

Device is a valid device enclosed in quotation marks that specifies on which *device* you want to set the width. See Chapter 1 for valid device names.

When you specify *device*, BASIC stores the new width and does not change the current width of the device. When a subsequent OPEN statement opens that device, BASIC uses the new width while the file is open. After you close the file, the device returns to the previous width.

When you set the width for the line printer or the communications channel, BASIC sends a carriage return after every *size* character. Specifying a width of 255 disables line wrapping. Doing this is the same as specifying an “infinite” width. WIDTH 255 is the default for the communications channel. For example:

```
10 WIDTH LPRINT 100
20 LPRINT "This line is 100 characters long. See
   what happens when you print a string longer than
   width."
```

Line 10 sets the printer width to 100 characters. After printing 100 characters, BASIC issues a carriage return. The carriage return causes the printer to print the remaining characters on the next line.

To set WIDTH at the screen, you may omit the LPRINT option in the first form of the syntax, like this:

```
WIDTH 40
```

or you may use the third form of the syntax and specify the device:

```
WIDTH "SCRN:", 40
```

You may only use the WIDTH statement to select a width of 80 if you are using the VM-2 Monochrome Monitor or CM-2 Color Monitor. If you are using the VM-2 Monochrome Monitor or the CM-2 Color Monitor, you should note the following:

- If you change the screen width, BASIC clears the screen and sets the background to black and the foreground to white.
- Changing the screen width does not affect the color enabling/disabling value (burst value).
- If you are in Screen Mode 1, changing the WIDTH to 80 forces the screen into Screen Mode 2.
- If you are in Screen Mode 2, changing the WIDTH to 40 forces the screen into Screen Mode 1.

If you attempt to select a size outside the range 0 to 255, an Illegal function call error results.

Examples

```
WIDTH LPRINT 132
WIDTH "LPT1:", 132
```

Both these statements change the printer width to 132. The second statement does not change the printer width until LPT1: is specified as the device in an OPEN statement.

```
10 WIDTH LPRINT 80
.
.
100 OPEN "LPT1:" FOR OUTPUT AS #1
.
150 PRINT #1
.
1000 WIDTH #1, 40
```

Line 10 changes the width of the printer to 80 characters. Line 150 prints the records as 80 characters each. After BASIC executes Line 1000, Line 150 prints the records as 40 characters each.

WINDOW

Statement

WINDOW [**SCREEN**] [(*x1,y1*)-(*x2,y2*)]

Lets you change the physical coordinates of the screen (or current viewport) by defining *world coordinates*. World coordinates can be any single-precision floating point numbers, including numbers outside the physical range of the screen as defined by the VIEW statement.

Note: The viewport is set to the entire screen by default. For more information on viewports, see the VIEW command.

(*x1,y1*) specifies the world coordinates for the upper-left corner of the screen. *x* is the horizontal coordinate, and *y* is the vertical coordinate.

(*x2,y2*) specifies the world coordinates for the lower-left corner of the display. *x* is the horizontal coordinate, and *y* is the vertical coordinate.

The SCREEN option tells BASIC to set the coordinates like the screen display where the lesser y-coordinate is in the upper-left corner of the screen. If you omit screen, BASIC inverts the y-coordinates to show a true cartesian coordinate system. That is, the lesser y-coordinate is in the lower-left corner of the screen.

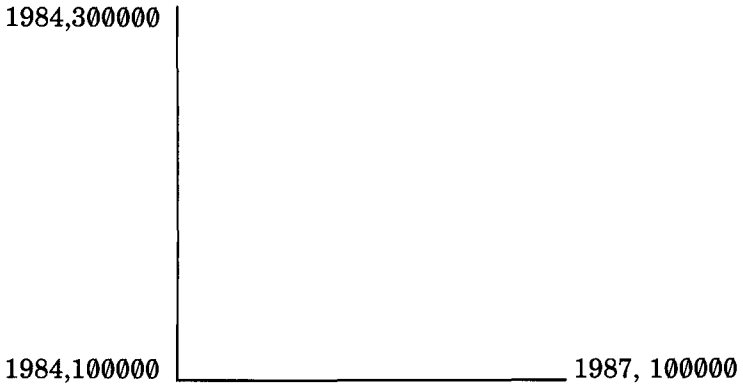
WINDOW lets you plot points outside the normal screen coordinate limits by setting new world coordinates to the screen. WINDOW transforms the new world coordinates onto the screen, usually altering the aspect ratio.

Note: CIRCLE, GET, and PUT do not use world coordinates.

You can easily plot graphs by specifying coordinates that are directly proportional to the limits of the graph. For example, to plot the increase of sales from 1984 to 1987 with sales averaging 100,000 to 300,000, you can use the following command:

```
WINDOW (1984,100000)-(1987,300000)
```

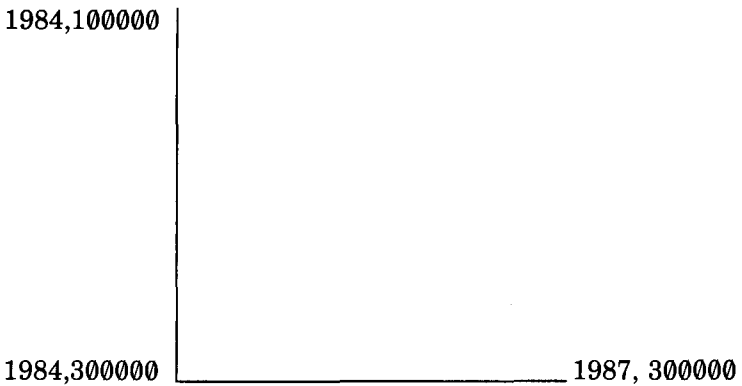

The coordinates can be pictured for commands that use world coordinates:



If you give the command:

`WINDOW SCREEN (1984,100000)-(1987,300000)`

the coordinates can be pictured as follows for commands that use world coordinates:



Note: RUN, SCREEN and WINDOW statements, without parameters, define the entire screen as the window.

WRITE

Statement

WRITE *data*[,*data*,...]

Writes data to the screen.

Data can be any string or numeric expression or variables. If you omit *data*, BASIC outputs a blank line.

The only difference between WRITE and PRINT is that WRITE prints commas between the data items and prints quotation marks around strings.

WRITE#**Statement****WRITE#***buffer, data[,data,...]*

Writes *data* to a sequential access disk file.

Buffer is the number assigned to the file when you opened it.

Data may be numeric or string expressions. If you specify more than one data item, separate the items with commas.

WRITE# inserts commas between the data items it writes to disk. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

WRITE# inserts a carriage return after writing the last data item to disk.

Example

```
A$="MICROCOMPUTER": B$="NEWS"  
WRITE#1, A$,B$
```

writes the following image to disk:

```
"MICROCOMPUTER","NEWS"
```


TECHNICAL INFORMATION

This chapter provides various information of a technical nature. If you are just beginning to use BASIC, you may want to skip this chapter until later.

Interfacing with Assembly-Language Subroutines

This section is for users who call subroutines written in other languages from their BASIC programs. BASIC provides for interfacing with subroutines through the `USR` function and through the `CALL` and the `CALLS` statements.

You can load your assembly language subroutine into BASIC's work area or into another segment of memory. We will show you both methods.

Memory Allocation Outside BASIC's Work Area

When you load BASIC, the `DS` (data segment) register is set to the address of BASIC's work area. To access an area of memory outside this work area, execute a `DEF SEG` statement to specify the address of the segment of memory you are accessing. If you don't execute a `DEF SEG` statement, your `CALL`, `CALLS`, or `USR` statements transfer control to an area within BASIC's work area. After returning from the subroutine, execute another `DEF SEG` statement to restore the `DS` register to its original value. See `DEF SEG` in Chapter 10 for more information.

Memory Allocation Inside BASIC's Work Area

To set aside memory space for an assembly language subroutine within BASIC's work area, use the `/M:` switch when you load BASIC. See Chapter 2 for a review of the start-up procedure.

The /M: switch sets the highest memory address that BASIC can use. The value that you specify with the /M: switch tells BASIC that it can use all memory up to that offset. Load your subroutine at that offset. Using the /M: switch prevents BASIC from destroying your subroutine. For example,

```
BASIC /M:&HF000
```

sets the highest memory location that BASIC can use at hexadecimal address EFFF. This reserves the highest 4K bytes of memory for your subroutine. You can load your subroutine at hexadecimal address &HF000 like this:

```
BLOAD "SUBA.ASM",&HF000
```

Note: For BASIC to BLOAD any subroutine, the subroutine must have a special 7-byte header that contains the necessary loading information. (If the subroutine does not have this header, BASIC returns a Bad file mode error.)

When you BSAVE a subroutine written in BASIC, the header is created automatically. But it does not exist on subroutines written in MS-FORTRAN, C, or assembly language. Therefore, to interface such subroutines with a BASIC program, you must either create the header manually or convert the subroutine to a format acceptable to BASIC. See "Converting Subroutines," below, for a breakdown of the header's contents and instructions on converting subroutines.

Stack Space

If you need more stack space when you call an assembly language subroutine, you can save the BASIC stack and set up a new stack for the subroutine. You must restore the BASIC stack before returning from the subroutine. You save the stack, create a new stack, and restore the stack in your subroutine.

Loading the Subroutine into Memory

You can use the operating system or the POKE statement to load the subroutine into memory. You may assemble the routines with the Macro Assembler (available through your Radio Shack dealer) and link them with Linker. The Linker is part of the MS-DOS package. To load the program file, observe these guidelines:

- Be sure that the subroutines do not contain any long references.
- Skip the first 512 bytes of the LINK output file and then read in the rest of the file.

Poking a Subroutine into Memory

You can code short subroutines in machine language and use the POKE statement to put the code into memory. To do so, follow these steps:

1. Code the machine language instructions for your subroutine.
2. Put the assembly instruction code for each byte of the machine language code into DATA statements, preceded by the &H symbols to denote that they are hexadecimal values.
3. Execute a loop that reads the DATA statements and POKEs them into an area of memory.

For example, the instruction code for the statement

```
PUSH    BP
```

is 55. The DATA statement for that instruction is

```
DATA    &H55
```

After the loop is complete, the subroutine is in memory. Whether you are using the USR function or the CALL statement to call the subroutine, you must set the value of the subroutine entry point as the location specified in the first POKE statement.

Converting Subroutines

The special BSAVE header mentioned earlier consists of:

Byte 1:	FD (specifies that the file is for BASIC's use)
Bytes 2 and 3:	<i>segment address</i>
Bytes 4 and 5:	<i>offset into segment address</i>
Bytes 6 and 7:	<i>length of machine language program (in bytes)</i>

The program follows the header. The last byte must be 1A (decimal 26, Control-Z, EOF).

To convert a program that does not have the header, first use the linker to create an executable (.exe) file from the machine language program. Then, via a simple BASIC program:

1. Open and read the file, ignoring (skipping) the first 512 bytes.
2. Continue reading the file, and poke each byte into its proper place in memory. Remember to use the DEF SEG command, if required.
3. When finished, close the file.
4. Use BSAVE to save the machine language object program on disk. The header is created automatically.
5. Use the BLOAD command to load the subroutine as you wish.

Example

You can use the following program to prepare a binary (.bin) file from an executable (.exe) file:

```
10 REM WHEN YOU ENTER BASIC, USE "/M:&HF000" TO
   PROTECT THE
20 REM TOP 4K BYTES OF MEMORY.
30 REM
40 OPEN "mprogram.exe" AS 1 LEN=1 : REM THE
   FILE TO BE CONVERTED
50 FIELD 1, 1 AS A$
60 REM
70 I = 512 : REM SKIP THE FIRST 512 BYTES OF
   THE ".exe" FILE
80 REM
90 ADDR%=&HF000: REM WHERE ROUTINE IS TO
   RESIDE AT TOP OF MEMORY
100 SIZ=1: REM WHERE NUMBER OF BYTES FOR "BSAVE"
    IS KEPT
110 REM
120 WHILE NOT (EOF(1))
130   I=I+1
140   GET 1,I
150   BYTE=ASC(A$) : REM CONVERT STRING
       CHARACTER INTO NUMBER
160   POKE ADDR%,BYTE: REM PUT THE BYTE WHERE YOU
       WANT IT
170   SIZ=SI+1
180   ADDR%=ADDR%+1: REM INCREMENT ADDRESS
       POINTER
190 WEND
200 CLOSE 1
210 REM
220 BSAVE "mprogram.bin",&HF000,SIZ-1
230 REM
240 END
```

CALL Statement

When the CALL statement is executed, the following occurs:

1. For each parameter in the parameter list, the two-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack. If the parameter is a string variable, the offset points to the *string descriptor*. See the section "Accessing String Parameters" in this appendix.

2. The BASIC return address code segment (CS) and offset (IP) are pushed onto the stack.
3. Control is transferred to the subroutine by an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in *variable*.

When the CALL statement is executed, the operating system loads the CS (code segment) register with the value specified in the last DEF SEG statement. If you are CALLing a subroutine within BASIC's work area, and no DEF SEG is required, the CS register is loaded with the address of BASIC's work area. This address is shifted left 4 bits; in other words, which is the same as multiplying it by 16 decimal (10 hexadecimal). Then, the offset of the subroutine is added to the segment address.

Example

$$17100 + 0020 = 17120$$

17120 is the absolute address of the first instruction in the subroutine.

Technical Functions

The called routine may destroy the previous contents of all registers. If you want to save the contents of the registers, the first instructions in the subroutine must be a PUSH for each register, and the last instructions in the subroutine must be a POP to restore the registers to their original value. You must execute a POP for every PUSH to maintain stack integrity.

The subroutine may refer to the passed parameters as positive offsets to the Base Pointer (BP). The CALLED routine must PUSH BP on the stack and then move the current stack pointer into BP. BP should be the first register you PUSH so that the parameters may be referenced as an offset to BP. The first 4 bytes of the stack contain the IP and CS register values that BASIC saves when the CALL is executed. To calculate the parameters offset from the BP, use this equation:

$$2 \cdot (\text{total parameters} - \text{parameter position}) + 6 = \text{offset}$$

For example, the address of parameter 1 is at 10(BP), parameter 2 is at 8(BP), and parameter 3 is at 6(BP).

Exiting the Subroutine

The called routine must execute a `RET number` statement to adjust the stack to the start of the calling sequence. The value of *number* is 2 times the number of parameters in the parameter list.

CALLS Statement

The `CALLS` statement is the same as `CALL` except the arguments are passed as segmented addresses. `CALLS` should be used to access MS-FORTRAN routines.

Because MS-FORTRAN routines need to know the segment value for each argument passed, the segment is pushed first, followed by the offset. `CALLS` pushes 4 bytes for each argument; therefore, the number in the `RET` statement (`RETn`) must be 4 times the number of arguments.

USR Function

When the `USR` statement is executed, the operating system loads the CS (code segment) register with the value specified in the last `DEF SEG` statement. If you are accessing a subroutine within BASIC's work area and no `DEF SEG` is required, the CS register is loaded with the address of BASIC's work area. This address is shifted left 4 bits; which is the same as multiplying it by 16 decimal (10 hexadecimal). Then the offset of the subroutine is added to the segment address.

Example

$$17100 + 0020 = 17120$$

This is the absolute address of the first instruction in the subroutine.

Technical Functions

When the USR function call is made, register AL contains a value that specifies the type of argument that was given. The value in AL may be one of the following:

Value in AL	Type of Argument
2	2-byte integer (two's complement)
3	String
4	Single precision floating-point number
8	Double precision floating-point number

If the argument is a string, the DX register pair points to the "string descriptor." See the section "Accessing String Variables" in this chapter.

If the argument is a number, the BX register pair points as follows:

- To the least significant byte (for an integer)
- To the least significant mantissa (for a single precision real number)
- To the FAC-3 location (for a double precision real number).

See the chart that follows:

Relative	FAC	Integer	SNG real	DBL real
&H0000	FAC-7			LS mantissa
&H0001	FAC-6			mantissa
&H0002	FAC-5			mantissa
&H0003	FAC-4			mantissa
&H0004	FAC-3	lower byte	LS mantissa	mantissa
&H0005	FAC-2	higher byte	mantissa	mantissa
&H0006	FAC-1		MS mantissa	MS mantissa
&H0007	FAC-0		exponent	exponent

LS and MS stand for "least significant" and "most significant."

Exiting the Subroutine

The subroutine must execute a RET 7 statement to adjust the stack to the start of the calling sequence.

How Variables are Stored

BASIC stores variables in its data segment as follows:

Byte	Contents	Description
Byte 0	Type	Identifies the type of variable stored at this location: 2 integer 3 string 4 single precision 8 double precision
Bytes 1 and 2	Name	The first 2 characters of the variable name.
Byte 3	Integer 3 - 38	Integer is the number of additional characters in the variable name.
Byte 4 + integer stored in Byte 3	Name	The remainder of the variable name is stored at bytes 4 + the integer stored in Byte 3.
Byte 4 + length	Data	The contents of the variable are stored in the bytes immediately following the variable name. The data can be 2, 3, 4, or 8 bytes in length, depending on the type of data.

At least 3 bytes are required to store any variable name. A 1- or 2-character variable name occupies exactly 3 bytes. Bytes 1 and 2 for the first 2 characters and Byte 3 to contain a zero to indicate that there are no additional characters in the variable name. If the variable name only contains 1 or 2 characters, the data is stored beginning at Byte 4. As you can see, the location of the first actual byte of data depends on the length of the variable name. VARPTR returns the offset of the first actual byte of data, not the offset of the beginning of the storage area.

Accessing String Variables

If the parameter passed in a CALL statement is a string expression, the parameters offset points to the *string descriptor*. If the argument passed in a USR function call is a string expression, the DX register points to the *string descriptor*.

The *string descriptor* is a 3-byte area of memory that points to the text of the string. The *string descriptor* contains the following:

Byte 0 contains the length of the string (0 to 255).

Byte 1 contains the lower 8 bits of the string starting address in BASIC's data segment.

Byte 2 contains the upper 8 bits of the string starting address in BASIC's data segment.

The text of the string may be altered by the subroutine, but the length of the string **must not** be changed. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Since the *string descriptor* points to an area of memory in your BASIC program, you must be careful not to alter or destroy your program. To avoid unpredictable results, add the concatenation symbol (+) to the string. This forces the string to be copied into string space, where the string may be modified without affecting the program.

Example

```
20 A$ = "MONTHLY SALES REPORT" + " "
```

File Control Block

A file control block is a storage area in BASIC's data segment that contains information BASIC needs for all functions performed on that file. When you execute the VARPTR function and specify the buffer number, BASIC returns the address of the BASIC file control block for that file. Note that this is the BASIC file control block, not the DOS file control block. The address is specified as an offset into BASIC data segment. In this section we define the information in the file control block. Offsets are relative to the value returned by VARPTR. Length is in bytes.

OFFSET LENGTH DESCRIPTION

0	1	Mode	The mode in which the file was opened: 1 - Input Only 2 - Output Only 4 - Random I/O 16 - Append Only
1	38	FCB	Disk File Control Block. Refer to DOS User's Guide for contents.
39	2	CURLOC	Number of sectors read or written for sequential access. For random access, it contains the last record number + 1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in Input buffer.
43	3	***	Reserved for future expansion.
46	1	DEVICE	Device number: 0-9 - Disks A: thru J: 255 - KYBD: 254 - SCRNL: 253 - LPT1: 251 - COM1: 250 - COM2: (not available) 249 - LPT2: (not available) 248 - LPT3: (not available)
47	1	WIDTH	Device width.
48	1	POS	Position in buffer for PRINT#.

49	1	FLAGS	Internal use during LOAD/SAVE; not used for data files.
50	1	OUTPOS	Output position used during tab expansion.
51	128	BUFFER	Physical data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in Sequential I/O mode.
179	2	VRECL	Variable length record size. Default is 128. Set by length option in OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk files only. Output position for PRINT#, INPUT#, and WRITE#.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine File data in Random I/O mode.

User Installed Devices

When writing device drives to use with BASIC, note the following rules:

- BASIC sends only a carriage return as an end of line. If the device requires a line feed also, you must provide for this in your driver.
- BASIC must read and write control information to the device. Reading and writing Device Control data is handled by the BASIC IOCTL statement and the IOCTL\$ function.
- Your driver must provide the following control functions:

The driver must set a maximum line width as requested by the OPEN statement.

The driver must return the current maximum line width when BASIC asks for it.

Input Devices must return an “end-of-file” condition to BASIC if you want to be able to close sequential input files open to the device driver. This is used by the EOF statement.

Input Devices should return a ^Z CTRL Z if BASIC attempts to read past the end of the device input stream. BASIC uses this to give an “Input past end” error.

For more information on device drives, see the *Programmer's Reference* manual for your computer. It is available at your Radio Shack Computer Store.

Information for Creating Child Processes

When writing programs for use as child processes from BASIC, please note the following rules and information:

- Child processes that use the screen device might modify the screen mode parameters. If necessary, restore these parameters from BIOS.
- Save and restore interrupt vectors the child process uses.
- BASIC places many hardware devices in specific states. These devices include an Interrupt Controller, Counter Timers, DMA Controller, I/O Latch, and Uarts.

- Be careful when altering any files opened by the BASIC parent program. The BASIC parent program should close all files before executing SHELL and then reopen them upon return.
- When the SHELL command executes, BASIC tries to free any memory not being used. However, BASIC does not free memory preserved with the /M switch. This may cause an Out of memory error.

To avoid this, load your machine language routines before entering BASIC. Use Interrupt 27 to let the routines to exit MS-DOS but still remain in memory.

For more information, see the *Programmer's Reference* manual for your computer (sold separately).

- Never use Interrupt 27 on a child process. If you attempt to terminate a child process but have it remain in memory, BASIC may not have enough room to expand its workspace to its original size. If the workspace cannot be restored, BASIC closes all files, prints the error message SHELL can't continue, and exits to MS-DOS.
- You cannot run BASIC as a child process to itself.

BASIC ERROR CODES AND MESSAGES

Number	Message
---------------	----------------

1	NEXT without FOR
----------	-------------------------

BASIC executed a NEXT statement without previously executing a FOR statement, or a variable in a NEXT statement does not correspond to a previously executed FOR statement.

2	Syntax error
----------	---------------------

BASIC encountered a line that contains an incorrect sequence of characters (such as unmatched parentheses, misspelled statement, incorrect punctuation, and so on). BASIC automatically enters the edit mode at the line that caused the error.

3	RETURN without GOSUB
----------	-----------------------------

BASIC executed a RETURN statement without previously executing a GOSUB statement.

4	Out of DATA
----------	--------------------

When executing a READ statement, BASIC could not find any DATA statements or unread data items.

5	Illegal function call
----------	------------------------------

A parameter that is out of range was passed to a math or string function. This error may also occur as the result of:

- negative array subscript or an unreasonably large array subscript.
- negative or zero argument with LOG.
- negative argument to SQR.
- negative mantissa with a noninteger exponent.
- invalid exponential number.

Number Message

- a call to a USR function without a starting address set by DEF USR.
- improper argument to MID\$, LEFT\$, RIGHT\$, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or LEFT\$, RIGHT\$, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO.
- negative record number used with GET or PUT.

6 Overflow

The result of a calculation was too large to be represented in BASIC numeric format. If underflow occurs, the result is zero, and execution continues without an error.

7 Out of memory

A program is too large, has too many FOR loops or GOSUBs, has too many variables, or has expressions that are too complicated.

8 Undefined line number

A nonexistent line was referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.

9 Subscript out of range

An array element is referenced with a subscript outside the dimensions of the array or with the wrong number of subscripts.

10 Redimensioned Array/Duplicate Definition

BASIC encountered 2 DIM statements for the same array, or a DIM statement after the default dimension of 10 had already been established for that array.

Number	Message
11	Division by zero An expression includes division by zero, or the operation of involution results in zero being raised to a negative power. BASIC supplies machine infinity with the sign of the numerator as the result of the division, or it supplies positive machine infinity as the result of the involution. Execution then continues.
12	Illegal direct A statement that is illegal as a command was entered at BASIC's prompt.
13	Type mismatch A string variable name was assigned a numeric value or vice versa. A string function was given a numeric argument or vice versa.
14	Out of string space The amount of memory used by string variables exceeded the amount of free memory.
15	String too long An attempt was made to create a string more than 255 characters.
16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.
17	Can't continue An attempt was made to continue a program that: <ul style="list-style-type: none">● halted because of an error.● was modified during a break in execution.● does not exist.

Number	Message
18	Undefined user function. A USR function was called before providing a function definition (DEF USR statement).
19	No RESUME BASIC executed an error-handling routine that did not have a RESUME statement.
20	RESUME without error BASIC executed a RESUME statement when no error had occurred.
21	Unprintable error An error message is not available for the error that occurred.
22	Missing operand BASIC encountered an expression that contained an operator but no operand.
23	Line buffer overflow The line being input is too long.
24	Device Timeout BASIC did not receive information from an I/O device within a predetermined amount of time.
25	Device Fault An incorrect device designation has been entered.
26	FOR without NEXT BASIC executed a FOR statement that did not have a matching NEXT.
27	Out of Paper BASIC received an out of paper status from the printer.

Number	Message
29	WHILE without WEND BASIC encountered a WHILE statement that did not have a matching WEND.
30	WEND without WHILE BASIC executed a WEND statement before executing a WHILE statement.

Disk Errors

Number	Message
50	FIELD overflow A FIELD statement is allocating more bytes than the specified record length of the direct access file.
51	Internal error An internal malfunction has occurred in BASIC. Report to Radio Shack the conditions under which the message appeared.
52	Bad file number BASIC encountered a reference to a buffer number that is not open or is out of the range of the number of files specified when BASIC was loaded.
53	File not found A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.
54	Bad file mode An attempt was made to use PUT, GET, or LOF with a sequential file, to LOAD a direct file, or to execute an OPEN statement with a file mode other than I, O, R, E or D.
55	File already open BASIC encountered an OPEN statement for sequential output, or a KILL statement, for a file that is already open.
57	Device I/O Error An Input/Output error occurred. This is a fatal error; the operating system cannot recover it.
58	File already exists The filename specified in a NAME statement is identical to a filespec already in use on the disk.

Number	Message
61	Disk full All disk storage space is in use.
62	Input past end BASIC executed an INPUT statement after all the data in the file had been read, or BASIC executed an INPUT statement to a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	Bad record number In a GET or PUT statement, the record number is either greater than the maximum allowed (16,777,215) or equal to zero.
64	Bad file name An illegal pathname was used with a LOAD, SAVE, KILL, or OPEN statement (for example, a filename with too many characters).
66	Direct statement in file Information in a non-ASCII format was encountered while LOADING an ASCII-format file. The LOAD is terminated.
67	Too many files The diskette already contains the maximum number of files allowed. This usually occurs on SAVE or OPEN. An attempt was made to create a new file (using SAVE or OPEN) when all directory entries are full.
68	Device Unavailable An attempt was made to open a file to a non-existent device. It may be that hardware does not exist to support the device, such as LPT2: or LPT3:, or that the device is disabled. This occurs if an OPEN "COM1:... statement is executed but the user disabled RS232 support via the /C:0 switch directive on the command line.

Number	Message
69	Communication buffer overflow Not enough space has been reserved for the communications buffer.
70	Disk write protected Occurs when an attempt is made to write to a diskette that is write-protected. Use an ON ERROR GOTO statement to detect this situation and request user action.
71	Disk not Ready Occurs when the diskette drive door is open or a diskette is not in the drive. Use an ON ERROR GOTO statement to recover.
72	Disk media error Occurs when the FDC controller detects a hardware or media fault. This usually indicates harmed media. Copy any existing files to a new diskette and re-format the damaged diskette. FORMAT flags any bad tracks and records them in a special file. The remainder of the diskette is then usable.
73	Advanced Feature
74	Rename across disks An attempt was made to rename a file with a new drive designation. BASIC does not allow this.
75	Path/File Access Error During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct Path-to-Filename connection.
76	Path not found The OPEN, MKDIR, CHDIR, or RMDIR statement references a path that does not exist.
77	Deadlock

BASIC Reserved Words and Derived Functions

Reserved BASIC Words

ABS	DELETE	IOCTL	OPTION	SIN
AND	DIM	IOCTL\$	OR	SOUND
ASC	DRAW	KEY	OUT	SPACE\$
ATN	EDIT	KEY\$	PAINT	SPC(
AUTO	ELSE	KILL	PALETTE	SQR
BEEP	END	LEFT\$	PALETTE USING	STEP
BLOAD	ENVIRON	LEN	PCOPY	STICK
BSAVE	ENVIRON\$	LET	PEEK	STOP
CALL	EOF	LINE	PEN	STR\$
CDBL	EQV	LIST	PLAY	STRIG
CHAIN	ERASE	LLIST	PMAP	STRING\$
CHDIR	ERDEV	LOAD	POINT	SWAP
CHR\$	ERDEV\$	LOC	POKE	SYSTEM
CINT	ERL	LOCATE	POS	TAB(
CIRCLE	ERR	LOCK	PRESET	TAN
CLEAR	ERROR	LOF	PRINT	TERM
CLOSE	EXP	LOG	PRINT#	THEN
CLS	FIELD	LPOS	PSET	TIME\$
COLOR	FILES	LPRINT	PUT	TIMER
COM	FIX	LSET	RANDOMIZE	TO
COMMON	FN	MERGE	READ	TROFF
CONT	FOR	MID\$	REM	TRON
COS	FRE	MKDIR	RENUM	UNLOCK
CSRLIN	GET	MKD\$	RESET	USING
CSNG	GOSUB	MKI\$	RESTORE	USR
CVD	GOTO	MKS\$	RESUME	VAL
CVI	HEX\$	MOD	RETURN	VARPTR
CVS	IF	MOTOR	RIGHT\$	VARPTR\$
DATA	IMP	NAME	RMDIR	VIEW
DATE\$	INKEY\$	NEW	RND	WAIT
DEF	INP	NEXT	RSET	WEND
DEFDBL	INPUT	NOISE	RUN	WHILE
DEFINT	INPUT#	NOT	SAVE	WIDTH
DEFSNG	INPUT\$	OCT\$	SBN	WINDOW
DEFSTR	INSTR	OFF	SCREEN	WRITE
DEF FN	INT	ON	SGN	WRITE#
DEF USR	INTER\$	OPEN	SHELL	XOR

Derived BASIC Functions

Functions which are not intrinsic to BASIC may be calculated as follows:

Function	BASIC Equivalent
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
CONTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/$ $\text{SQR}(-X \cdot X + 1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/$ $\text{SQR}(-X \cdot X + 1)) + 1.5708$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/$ $\text{SQR}(X \cdot X - 1)) + (\text{SGN}(X) - 1)$ $\cdot 1.5708$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X \cdot X - 1))$ $+ (\text{SGN}(X) - 1) \cdot 1.5708$
INVERSE COTANGENT	$\text{ARCCOT}(X) = \text{ATN}(X) + 1.5708$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/$ $(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = (\text{EXP}(X) + (\text{EXP}(-X))/$ $(\text{EXP}(X) - \text{EXP}(-X))$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X$ $+ \text{SQR}(X \cdot X + 1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X$ $+ \text{SQR}(X \cdot X - 1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1 + X)/(1 - X))/$ 2

Function	BASIC Equivalent
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X * X + 1) + 1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X) * \text{SQR}(X * X + 1) + 1)/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X + 1)/(X - 1))/2$

KEYBOARD AND CHARACTER CODE CHARTS

Keyboard ASCII/Scan Codes

The following table lists the keys, in scan code order, and the ASCII codes generated by each (which depends on the shift status). The entries in the table are:

- **SCAN CODE** — A value in the range 01H-54H (hexadecimal) that uniquely describes which key is pressed.
- **NORM** — The normal (unshifted) ASCII value (returned when only the indicated key is pressed).
- **UPPERCASE** — The shifted ASCII value (returned when **SHIFT** is also pressed).
- **CTRL** — The control ASCII value (returned when **CTRL** is also pressed.)
- **ALT** — The alternate ASCII value (returned when **ALT** is also pressed).

All numeric values in the table are expressed in hexadecimal. Those values preceded by an X are extended ASCII codes. (They are preceded by an ASCII NUL [=00].)

The following notations are used in the table:

- indicates that no ASCII code is generated.
- ** indicates that no ASCII code is generated, and that, instead, the special function described later is performed.
- † indicates that you can generate the ASCII codes of decimal numbers in the range 1 to 255. Hold down **ALT** while you type *on the numeric keypad* any decimal number in the acceptable range. When you release **ALT**, the ASCII code of the number is generated and displayed.

Note: When the NUM LOCK light is on, the BASE CASE characters are produced.

Appendix B

Scan code to ASCII translations for US English Tandy 3000 keyboard

key # - SCAN CODE	NORM CASE (ASCII code)		UPPER CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
01	ESC	1B	ESC	1B	ESC	1B	—	—
02	1	31	!	21	—	—	ALT1	X078
03	2	32	@	40	NULL	00	ALT2	X079
04	3	33	#	23	—	—	ALT3	X07A
05	4	34	\$	24	—	—	ALT4	X07B
06	5	35	%	25	—	—	ALT5	X07C
07	6	36	^	5E	RS	1E	ALT6	X07D
08	7	37	&	26	—	—	ALT7	X07E
09	8	38	*	2A	—	—	ALT8	X07F
0A	9	39	(28	—	—	ALT9	X080
0B	0	30)	29	—	—	ALT0	X081
0C	-	2D	—	5F	US	1F	ALT-	X082
0D	=	3D	+	2B	—	—	ALT=	X083
0E	BS	08	BS	08	DEL	7F	—	—
0F	→	09	←	X00F	—	—	—	—
10	q	71	Q	51	DC1	11	ALTQ	X010
11	w	77	W	57	ETB	17	ALTW	X011
12	e	65	E	45	ENQ	05	ALTE	X012
13	r	72	R	52	DC2	12	ALTR	X013
14	t	74	T	54	DC4	14	ALTT	X014
15	y	79	Y	59	EM	19	ALTY	X015
16	u	75	U	55	NAK	15	ALTU	X016
17	i	69	I	49	HT	09	ALTI	X017
18	o	6F	O	4F	SI	0F	ALTO	X018
19	p	70	P	50	DLE	10	ALTP	X019
1A	[5B	{	7B	ESC	1B	—	—
1B]	5D	}	7D	GS	1D	—	—
1C	CR	0D	CR	0D	LF	0A	—	—
1D	CTRL	—	CTRL	—	CTRL	—	CTRL	—
1E	a	61	A	41	SOH	01	ALTA	X01E
1F	s	73	S	53	DC3	13	ALTS	X01F
20	d	64	D	44	EOT	04	ALTD	X020
21	f	66	F	46	ACK	06	ALTf	X021
22	g	67	G	47	BEL	07	ALTg	X022
23	h	68	H	48	BS	08	ALTH	X023
24	j	6A	J	4A	LF	0A	ALTj	X024
25	k	6B	K	4B	VT	0B	ALTk	X025
26	l	6C	L	4C	FF	0C	ALTl	X026
27	;	3B	:	3A	—	—	—	—
28	'	27	"	22	—	—	—	—
29	`	60	~	7E	—	—	—	—
2A	left SHIFT	—	left SHIFT	—	left SHIFT	—	left SHIFT	—
2B	\	5C		7C	FS	1C	—	—
2C	z	7A	Z	5A	SUB	1A	ALTz	X02C
2D	x	78	X	58	CAN	18	ALTx	X02D
2E	c	63	C	43	ETX	03	ALTC	X02E
2F	v	76	V	56	SYN	16	ALTv	X02F
30	b	62	B	42	STX	02	ALTB	X030
31	n	6E	N	4E	SO	0E	ALTn	X031
32	m	6D	M	4D	CR	0D	ALTM	X032

key # - SCAN CODE	NORM CASE (ASCII code)		UPPER CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
33	,	2C	<	3C	—	—	—	—
34	.	2E	>	3E	—	—	—	—
35	/	2F	?	3F	—	—	—	—
36	right SHIFT —		right SHIFT —		right SHIFT —		right SHIFT —	
37	*	2A	PrScr**		CPrScr** X072		—	
38	ALT —		ALT —		ALT —		ALT —	
39	SPACE 20		SPACE 20		SPACE 20		SPACE X020	
3A	CAPS —		CAPS —		—		CAPS —	
3B	F1	X03B	F11	X054	F21	X05E	F31	X068
3C	F2	X03C	F12	X055	F22	X05F	F32	X069
3D	F3	X03D	F13	X056	F23	X060	F33	X06A
3E	F4	X03E	F14	X057	F24	X061	F34	X06B
3F	F5	X03F	F15	X058	F25	X062	F35	X06C
40	F6	X040	F16	X059	F26	X063	F36	X06D
41	F7	X041	F17	X05A	F27	X064	F37	X06E
42	F8	X042	F18	X05B	F28	X065	F38	X06F
43	F9	X043	F19	X05C	F29	X066	F39	X070
44	F10	X044	F20	X05D	F30	X067	F40	X071
45	NUM LOCK —		NUM LOCK —		PAUSE **		NUM LOCK —	
46	SCROLL LOCK —		SCROLL LOCK —		BREAK **		SCROLL LOCK —	
54	SYS**		SYS**		SYS**		SYS**	

Numeric key pad

SCAN CODE	NUM LOCK (ASCII code)		BASE CASE (ASCII code)		CTRL CASE (ASCII code)		ALT CASE (ASCII code)	
47	7	37	HOME	X047	CLR SCN	X077	†	—
48	8	38	↑	X048	—	—	†	—
49	9	39	PG UP	X049	TOP OF TEXT AND HOME	X084	†	—
4A	—	2D	—	2D	—	—	—	—
4B	4	34	←	X04B	LEFT ONE WORD	X073	†	—
4C	5	35	—	—	—	—	†	—
4D	6	36	→	X04D	RIGHT ONE WORD	X074	†	—
4E	+	2B	+	2B	—	—	—	—
4F	1	31	END	X04F	ERASE TO EOL	X075	†	—
50	2	32	↓	X050	—	—	†	—
51	3	33	PG DN	X051	ERASE TO EOS	X076	†	—
52	0	30	INS	X052	—	—	†	—
53		2E	DEL	X053	—	—	—	—

ASCII and Scan Code Special Handling

CTRL BREAK BREAK	Empties the keyboard queue and executes the keyboard break interrupt (int 1BH). Places a NULL ASCII scan code in the keyboard queue.
CTRL S PAUSE	Delays system activity (except external interrupts) until you press another key.
SHIFT PR SCR PrScr	Invokes the BIOS print screen function (int 5H). A second PrScr halts the printer output.
CTRL PR SCR CPrScr	Tells MS-DOS to direct console output to both the printer and the console. A second CPrScr halts printer output.
SYS SYS	Invokes interrupt 15H (sequence write) and Register AX=8500H. When you release SYS , MS-DOS invokes interrupt 15H and Register AX=8501H.

ASCII Character Codes

The previous table listed the ASCII codes (in hexadecimal) generated by each key. The following table lists the characters generated by those ASCII codes. (Note: All ASCII codes in this table are expressed in **decimal** form.)

Note: All ASCII codes in this table are expressed in decimal form.

You can display the characters listed by doing either of the following:

- Using the BASIC statement `PRINT CHR$(code)`, where *code* is the ASCII code.
- Pressing **ALT** and, without releasing it, typing the ASCII code on the **numeric keypad**.

For Codes 0-31, the table also lists the standard interpretations. The interpretations are usually used for control functions or communications.

Note: The BASIC program editor has its own special interpretation of some codes and may not display the character listed.

ASCII CHARACTER CODES

ASCII Code	Character	Control Character
000	(null)	NUL
001	☺	SOH
002	☹	STX
003	♥	ETX
004	♦	EOT
005	♣	ENQ
006	●	ACK
007	(beep)	BEL
008	■	BS
009	(tab)	HT
010	(line feed)	LF
011	(home)	VT
012	(form feed)	FF
013	(carriage return)	CR
014	🎵	SO
015	⚙	SI
016	▶	DLE
017	◀	DC1
018	‡	DC2
019	!!	DC3
020	¶	DC4
021	§	NAK
022	—	SYN
023	‡	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESC
028	(cursor right)	FS
029	(cursor left)	GS
030	(cursor up)	RS
031	(cursor down)	US

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
032	(space)	068	D
033	!	069	E
034	''	070	F
035	#	071	G
036	\$	072	H
037	%	073	I
038	&	074	J
039	'	075	K
040	(076	L
041)	077	M
042	*	078	N
043	+	079	O
044	,	080	P
045	-	081	Q
046	.	082	R
047	/	083	S
048	0	084	T
049	1	085	U
050	2	086	V
051	3	087	W
052	4	088	X
053	5	089	Y
054	6	090	Z
055	7	091	[
056	8	092	\
057	9	093]
058	:	094	^
059	;	095	_
060	<	096	`
061	=	097	a
062	>	098	b
063	?	099	c
064	@	100	d
065	A	101	e
066	B	102	f
067	C	103	g

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
104	h	140	î
105	i	141	ì
106	j	142	Ä
107	k	143	Å
108	l	144	È
109	m	145	æ
110	n	146	Æ
111	o	147	ô
112	p	148	Ö
113	q	149	ò
114	r	150	û
115	s	151	ù
116	t	152	ÿ
117	u	153	Ö
118	v	154	Ü
119	w	155	¢
120	x	156	£
121	y	157	¥
122	z	158	Pt
123	{	159	ƒ
124		160	á
125	}	161	í
126	~	162	ó
127	☐	163	ú
128	Ç	164	ñ
129	ü	165	Ñ
130	é	166	à
131	â	167	o
132	ä	168	ç
133	ä	169	┌
134	ä	170	└
135	ç	171	½
136	ê	172	¼
137	ë	173	ï
138	è	174	«
139	ï	175	»

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
176	⌘	212	Ⓖ
177	⌘	213	Ⓕ
178	⌘	214	Ⓢ
179		215	Ⓢ
180	┐	216	Ⓢ
181	┐	217	┐
182	┐	218	┐
183	┐	219	■
184	┐	220	■
185	┐	221	■
186		222	■
187	┐	223	■
188	┐	224	α
189	┐	225	β
190	┐	226	┐
191	┐	227	π
192	┐	228	Σ
193	┐	229	σ
194	┐	230	μ
195	┐	231	τ
196	—	232	Φ
197	+	233	θ
198	┐	234	Ω
199	┐	235	δ
200	┐	236	∞
201	┐	237	Ø
202	┐	238	{
203	┐	239	∩
204	┐	240	≡
205	≡	241	±
206	┐	242	≥
207	┐	243	≤
208	┐	244	┐
209	┐	245	┐
210	┐	246	÷
211	┐	247	≈

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
248	°	252	η
249	●	253	²
250	•	254	■
251	√	255	(blank 'FF')

VIDEO DISPLAY WORKSHEET

The following page contains a video display worksheet of your text screen's coordinates. This map is provided to help you quickly position the cursor for screen prints. You'll find it especially useful for creating visually pleasing, easy-to-follow screen menus. See the CSRLIN and POS functions for information on returning the current cursor position. See the LOCATE statement and the TAB function for information on positioning the cursor.

EXTENDED CODES

For certain keys and key combinations, INKEY\$ returns a 2-character code. The first character is a null character (ASCII Code 00). The second is usually the scan code of the key(s) pressed. The key(s) and associated ASCII codes (in decimal) are listed below.

Second Character	Key(s) Pressed	Second Character	Key(s) Pressed
15	SHIFT TAB	64	F6
16	ALT Q	65	F7
17	ALT W	66	F8
18	ALT E	67	F9
19	ALT R	68	F10
20	ALT T	71	HOME
21	ALT Y	72	↑
22	ALT U	73	PG UP
23	ALT I	75	←
24	ALT O	77	→
25	ALT P	79	END
30	ALT A	81	PG DN
31	ALT S	82	INSERT
32	ALT D	83	DELETE
33	ALT F	84	SHIFT F1
34	ALT G	85	SHIFT F2
35	ALT H	86	SHIFT F3
36	ALT J	87	SHIFT F4
37	ALT K	88	SHIFT F5
38	ALT L	89	SHIFT F6
44	ALT Z	90	SHIFT F7
45	ALT X	91	SHIFT F8
46	ALT C	92	SHIFT F9
47	ALT V	93	SHIFT F10
48	ALT B	94	CTRL F1
49	ALT N	95	CTRL F2
50	ALT M	96	CTRL F3
59	F1	97	CTRL F4
60	F2	98	CTRL F5
61	F3	99	CTRL F6
62	F4	100	CTRL F7
63	F5	101	CTRL F8

Second Character	Key(s) Pressed	Second Character	Key(s) Pressed
102	CTRL F9	118	CTRL PG DN
103	CTRL F10	119	CTRL HOME
104	ALT F1	120	ALT 1
105	ALT F2	121	ALT 2
106	ALT F3	122	ALT 3
107	ALT F4	123	ALT 4
108	ALT F5	124	ALT 5
109	ALT F6	125	ALT 6
110	ALT F7	126	ALT 7
111	ALT F8	127	ALT 8
112	ALT F9	128	ALT 9
113	ALT F10	129	ALT 0
115	CTRL +	130	ALT -
116	CTRL →	131	ALT =
117	CTRL END	132	CTRL PG UP

INDEX

- ABS Fn 68, **71**
- Absolute coordinates 59
- Absolute value 71
- Active page 58, 272
- Addition 32
- ALT key 22
- AND 35
- Animation 143, 251
- Arctangent 73
- Arguments 2
- Arithmetic operators 31-32
- Arrays 39-43
 - defining 43, 116
 - erasing 126
 - types 42-43
- ASC Fn 68, **72**
- ASCII codes 72, 86, 271, 343-51
- Aspect ratio 56, 57
- Assembly-language subroutines
 - CALL 79, 321-22
 - calling 79, 321-22
 - converting 320-21
 - DEF USR 114
 - interfacing 317-18
 - loading 319
 - poking 319
 - USR 301, 323
- ATN Fn 68, **73**
- AUTO St 63, **74**

- Background colors 54, 55
- BASIC
 - concepts 23-37
 - derived function 340-41
 - device names 6
 - directory paths 4
 - disk files 3
 - editing 17-19
 - line numbers 23, 74, 258-59
 - loading 7-12
 - loading options 8-11

BASIC (cont.)
 pathnames 4
 program 23
 redirection 11-12
 reserved words 339
 sample session 13-15
 special keys 20-21
 statement 23
 wildcards 5-6
 work area 317-18
BEEP St 63, **75**
BLOAD St 63, **76-77**, 78
Boolean operators 34-36
Borders 54, 95, 97
Branching 200-13
BSAVE St 63, 76-77, **78**
Buffer 2, 133-34, 190, 214, 267, 310-11
Buffer, communications 9, 181, 186, 218-21, 250
Buffer, music 208-09
Buffer, printer 188

CALL St 63, **79**, 321-22
Calling subroutines 79, 321-22
CALLS St 63, **80**, 323
CDBL Fn 68, **81**
Chaining 82-84, 101
CHAIN St 63, **82-83**
CHDIR St 63, **85**
Child processes 275-76, 329-30
CHR\$ Fn 68, **86**
CINT Fn 68, **87**
CIRCLE St 63, **88-89**
CLOSE St 63, **93**, 260
Clear
 memory 91-92
 screen 94
CLEAR St 63, **91-92**
CLS St 63, **94**
Color 53-56, 95-96, 97-98
COLOR/Graphics St 55, 63, **95-96**
Color sets 54-55
COLOR/Text St 63, **97-98**
COM St 63, **99-100**
Comments 23, 257

- COMMON St 63, **101**
- Communications 124, 141, 181, 186, 218-21, 250
- Communications buffer 9, 181, 186, 218-21, 250
- Communications trapping 99-100, 200-01
- Compressed files 269
- Concatenation 32
- Concepts 23-37
- Constants 26-28
 - classifying 28
 - declaring 28
- CONT St 63, **102**, 285
- Converting precision 29-31, 81, 87, 104
- Converting strings 285, 302
- Coordinates
 - absolute 59
 - relative 59
 - physical 235, 236, 312-13
 - world 235, 236, 312-13
- COS Fn 68, **103**
- Cosine 103
- CSNG Fn 68, **104**
- CSRLIN Fn 68, **105**
- Current segment 113
- Cursor 105, 182, 238
- CVD Fn 68, **106**
- CVI Fn 68, **106**
- CVS Fn 68, **106**
- Data 24-26
 - constants 26-27
 - converting 29-31, 81, 87, 104, 196
 - double precision 25, 28, 29, 81, 106, 196
 - hexadecimal 25, 146
 - integers 24, 87, 106, 160, 196
 - manipulating 31
 - numeric 24-26
 - octal 26, 199
 - printing 189
 - single precision 25, 28, 29, 104, 196
 - strings 24, 29, 111, 158-59, 169, 170, 264
- DATA St 63, **107**
- Data files 91, 214-17, *see also* Disk files
- Date, retrieving 109-10
- Date, setting 109-10

- DATE\$ Fn 68, **109-10**
- Debugging 127, 316, 332
- DEFDBL St 63, **111**
- DEF FN St 63, **112**
- DEFINT St 63, **111**
- DEF SEG St 63, **113**
- DEFSNG St 63, **111**
- DEFSTR St 63, **111**
- DEF USR St 63, **114**
- DELETE St 63, **115**
- Derived functions 340-41
- Device errors 127, 128
- Device names 6
- Devices 6, 214-17, 310, 329
- DIM St 63, **116**
- Direct access 48-52, *see also Disk files*
- Directories 4
 - changing 85
 - creating 195
 - displaying 135
 - removing 265
- Directory path 4
- Disk files
 - buffer 2, 133-34, 190, 214, 267
 - CLOSE 93
 - closing 260
 - converting data 106
 - direct access 48-52
 - accessing 50-51, 140
 - closing 93
 - creating 49-50
 - EOF 124
 - FIELD 133-34
 - GET 140
 - locating records 182
 - LSET 190
 - MKD\$ 196
 - MKI\$ 196
 - MKS\$ 196
 - OPEN 214-17
 - PUT 249
 - RSET 267
 - displaying 135
 - end of file 124

Disk files (cont.)
 FIELD 133-34
 file control block 326-28
 length 185
 LOAD 179
 LOF 185
 MERGE 191
 OPEN 214-17
 renaming 197
 sequential access 45-48
 closing 93
 creating 45-47
 end of file 124
 EOF 124
 INPUT# 154-55
 INPUT\$ 156-57
 LINE INPUT# 176
 locating records 181
 OPEN 214-17
 PRINT# 245-47
 updating 47-48
Display page 58, 272
Division 32
 integer 32
Double precision 25
 CDBL 81
 CVD 106
 DEFDBL 111
 MKD\$ 196
DRAW St 63, **117-19**
Draw point 248

EDIT St 64, **120**
Editing 17-20, **120**
END St 64, **121**
End of file 124
ENVIRON St 64, **122**
ENVIRON\$ Fn 68, **123**
Environment String Table 122, 123
EOF Fn 68, **124**
Equal sign 33
EQV 35
ERASE St 64, **126**
ERDEV Fn 68, **127**

ERDEV\$ Fn 68, **128**
ERL St 64, **129**
ERR St 64, **130**
ERROR St 64, **131**
Error codes 331-38
Error messages 331-38
Errors 127-31, 331-38
 device 127-28
 ERDEV 127
 ERDEV\$ 128
 ERL 129
 ERR 130
 ERROR 131
 RESUME 262
 simulate 131
 trapping 129, 130, 208
EXP Fn 68, **132**
Exponent, natural 132
Exponential numbers 28
Exponentiation 32
Expressions 31, 147
Extensions 4

FIELD St 64, **133-34**
File control block 326-28
Filenames 4, 5
Files *see* *Disk files*
FILES St 64, **135**
FIX Fn 68, **136**
Formatting printing 189, 239-44
FOR/NEXT St 64, **137-38**
FORTRAN routines 80
FRE Fn 68, **139**
Function keys 163-64
 assigning 163
 displaying 164
 trapping 165-66, 205-06
Functions 37, 68-70
Function, user 112

GET St 64, 106, 140
GET/Communications St 64, **141**
GET/Graphics St 64, **142-43**
GOSUB St 64, **144**, 262

GOTO St 64, **145**
Graphics 53-60, 88-89, 117-19, 172-74, 224-26,
248, 251-53, 272-73, 305-06
Graphics modes 55
Greater Than sign 33
Greater Than/Equal To sign 33

HEX\$ Fn 68, **146**
Hexadecimal 1, 25, 146
Hierarchy of operators 36-37

IF/THEN/ELSE St 64, **147-48**
Image file 76-77, 78
IMP 35
Inequality sign 33
INKEY\$ Fn 68, **149-50**
INP Fn 68, **151**
INPUT St 64, **152-53**
INPUT# St 64, **154-55**
INPUT\$ St 64, **156-57**
Input
 communications 141
 device 154-55
 disk 140, 154-55, 156-57, 176
 graphics 142-43
 keyboard 149-50, 152-53, 156-57, 175
 joysticks 283, 287-88
 light pen 228
 memory 227
 music buffer 208-09
 port 151, 308
Input, redirection 9
INSTR Fn 68, **158-59**
INT Fn 68, **160**
Integer division 32
Integers 24, 28, 29
 CINT 87
 CVI 106
 DEFDBL 111
 FIX 136
 INT 160
 MKI\$ 196
IOCTL St 94, **161**
IOCTL\$ Fn 69, **162**

Joystick 283, 286, 287-88
trapping 210-11, 289-90

KEY St 64, **163-64**, 166
Keyboard codes 343-51
Keyboard input 149-50, 152-53, 156-57, 175
Keys 20-22, 165
Keys, user-defined 165
KEY/Trap St 64, **165-66**
Key trapping 165-66, 205-06
KILL St 64, **167**

LCOPY 64, **168**
LEFT\$ Fn 69, **169**
LEN Fn 69, **170**
Less Than sign 33
Less Than/Equal To sign 33
LET St 64, **171**
Light pen
trapping 207, 229
LINE St 64, **172-74**
LINE INPUT St 64, **175**
LINE INPUT# St 64, **176**
Line length 17
Line numbers 23
automatic 74
LIST St 65, **177**
LLIST St 65, **178**
LOAD St 14-15, 65, **179**
Loading
BASIC 7, 13
BASIC options 8-10
programs 14-15, 179
LOC Fn 69, **181**
LOCATE St 65, **182**
Locating cursor 105, 238
Locating record 181
LOC/Communication Fn 69, **181**
LOCK St 65, **183-84**
LOF Fn 69, **185**
LOF/Communication Fn 69, **186**
LOG Fn 69, **187**
Logarithms 132, 187

Logical operators 34-36

Loops 137-38

LPOS Fn 69, **188**

LPRINT St 65, **189**, 294

LSET St 65, **190**

Memory allocation 317-18

Memory image file 76-77, 78

Memory read 227

Memory size 91-92, 113, 139

MERGE St 65, **191**

MID\$ Fn 69, **194**

MID\$ St 65, **193**

MKD\$ Fn 69, **196**

MKDIR St 65, **195**

MKIS\$ Fn 69, **196**

MKS\$ Fn 69, **196**

MOD 32

Modulus arithmetic 32

MS-DOS 3, 275-76, 293

 child processes 275

 directory path 4

 directory structure 3

 names 5

 pathnames 4

 root 3

 SYSTEM 293

Multiplication 32

Music 230-34

 buffer 208-09

 PLAY 230-32

 trapping 208-09, 234

NAME St 65, **197**

Natural exponent 132

Natural logarithm 187

Negation 32

Nested loops 137-38

NEW St 65, **198**

NOT 35

Notations 1

Numbers

 converting 29-31, 81, 87, 104, 106

 double precision 25, 28, 29, 81, 106, 196

Numbers (cont.)

- hexadecimal 25, 146
- integers 24, 87, 106, 160, 196
- octal 26, 199
- single precision 25, 28, 29, 104, 196

Numeric constants 28

Numeric data 24-26

Numeric variables 28-29

OCT\$ Fn 69, **199**

Octal 1, 26, 199

ON COM()GOSUB St 65, 99-100, **200-01**

ON ERROR GOTO St 65, **202**, 262

ON/GOSUB St 65, **203**, 263

ON/GOTO St 65, **204**

ON KEY()GOSUB St 65, 166, **205-06**

ON PEN GOSUB St 65, **207**

ON PLAY()GOSUB St 65, **208-09**, 234

ON STRIG()GOSUB St 65, **210-11**, 286, 289-90

ON TIMER()GOSUB St 65, **212-13**, 299

OPEN St 65, **214-17**

OPEN"COM St 65, **218-21**

Operators

- arithmetic 31-32
- hierarchy 36
- logical 34-36
- relational 32-34
- string relational 33-34

OPTION BASE St 65, **222**

OR 35

OUT St 65, **223**

Output

- communication 250
- disk 245-47, 249, 315
- display 178, 239-40, 241-44, 248, 294, 314
- graphics 251-53
- memory 237
- music 230-32
- port 223
- printer 178, 189, 294
- sound 75, 91-92

Output redirection 11-12

Pages, video 58

PAINT St 66, **224-26**
Palettes 55, 95-96
Parameters 2
Pathnames 4
PEEK Fn 69, **227**, 237
PEN Fn 69, **228**
PEN/Trap St 65, 207, 228, **229**
Physical coordinates 235, 236, 312-13
PLAY Fn 69, **233**
PLAY St 66, **230-32**
PLAY/Trap St 65, 208-09, **234**
PMAP Fn 69, **235**
POINT Fn 69, **236**
POKE St 66, 227, **237**
Ports 151, 223, 308
POS Fn 69, **238**
Position cursor 182
Precision conversion 29-31, 81, 87, 104, 106
PRESET St 66, **248**
PRINT St 66, **239-40**, 294
PRINT USING St 66, **241-44**
PRINT# St 66, **245-47**
Print buffer 188
Printer 178, 188, 189, 294, 310-11
Printing, formatted 189, 239-44
Program
 elements 23
 execution 268
 line numbers 23, 258-59
 lines 23, 258-59
 listing 177, 178
 loops 137-38, 309
Program merging 191
Program renumbering 258-59
Program termination 121, 285
PSET St 66, **248**
PUT St 66, **249**
PUT/Communication St 66, **250**
PUT/Graphics St 66, **251-53**

RANDOMIZE St 66, **254**, 298
Random numbers 254, 266
READ St 66, 107, **255-56**, 261
Records 45

- Record size 9
- Redirection 9, 11-12
- Relational operators 32-34
 - with strings 33-34
- Relative coordinates 59
- REM St 66, **257**
- Remarks 66, **257**
- Removing directories 265
- Removing files 167
- Removing lines 115
- Removing programs 198
- Renaming files 197
- RENUM St 66, **258-59**
- Reserved words 339
- RESET St 66, **260**
- Resolution 56-57
- RESTORE St 66, **261**
- RESUME St 66, **262**
- Retrieving date 109-10
- Retrieving time 296-97
- RETURN St 66, **263**
- RIGHT\$ Fn 69, **264**
- RMDIR St 66, **265**
- RND Fn 69, **266**
- Root directory 3
- RSET St 66, **267**
- RUN St 13, 66, **268**

- Sample session 13-15
- SAVE St 14, 66, **269-70**
- Saving programs 14, 269-70
- Scan codes 343-46
- SCREEN Fn 69, **271**
- SCREEN St 66, **272-73**
- Screen, clear 94
- Screen modes 53-60, 272-73
- Search, strings 158-59
- Segment address 113
- Sequential access files 45-58
 - closing 93
 - creating 45-47
 - end of file 124
 - EOF 124
 - INPUT# 154-55

Sequential access files (cont.)
 INPUT\$ 156-57
 LINE INPUT# 176
 locating records 185
 OPEN 214-17
 PRINT# 245-47
 Updating 47-48
Setting date 109-10
Setting time 296-97
SGN Fn 69, **274**
SHELL St 67, **275-76**
SIN Fn 69, **277**
Sine 277
Single precision 25, 28, 29
 CSNG 104
 CVS 106
 DEFSNG 111
 MKSS 196
SOUND St 66, 75, 91-92, **278-79**
SPACES\$ Fn 69, **280**
SPC Fn 70, **281**
Speakers 75, 278-79
Special keys 20-21
SQR Fn 70, **282**
Square root 282
Stack space 91
Statements 23, 63-67
STICK Fn 70, **283**
STOP St 67, 102, **285**
STR\$ Fn 70, **285**, 302
STRIG Fn 70, 117, **287-88**
STRIG St 67, **286**
STRIG/Trap St 67, 210-11, **289-90**
STRING\$ Fn 70, **291**
String constants 26, 27
Strings 24, 26-27, 29, 111, 158-59, 169, 170, 264
String space 139
String variables 27, 29
Subroutines 79, 114, 144, 203, 263, 301
Subtraction 32
SWAP St 67, **292**
Syntax 2
SYSTEM St 67, **293**

TAB Fn 70, **294**
TAN Fn 70, **295**
Tangent 295
Terms 2, 62
Text mode 53, 54-55, 57, 58, 97-98, 272-73, 307
Tiling 225-26
Time 70
 setting 296-97
 trapping 298
TIMES Fn 70, **296-97**
TIMER Fn 70, **298**
TIMER/Trap St 67, 212-13, **299**
Tracer 300
Trapping
 communication 99-100, 200-01
 errors 130, 131, 202
 joystick 210-11, 289-90
 keys 165-66, 205-06
 light pen 207, 229
 music 208-09, 234
 timer 212-13, 299
TROFF St 67, **300**
TRON St 67, **300**
Typing programs 13-14

User installed devices 329
Unary minus 32
USR Fn 70, **301**, 323-24

VAL Fn 70, 285, **302**
Variables 27, 21, 171, 292, 303, 304, 325-26
 classifying 27, 29, 111
 clearing 91
 declaring 27-29, 111
 numeric 27, 111
 string 27, 111, 324
VARPTR Fn 70, **303**
VARPTR\$ Fn 70, **304**
Video aspect ratio 56-57
Video, clear 94
Video display worksheet 353
Video memory 58
Video pages 58

Video resolution 56-57

VIEW St 67, **305-06**

Viewports 305-06

VIEW PRINT St 67, **307**

WAIT St 67, **308**

WHILE/WEND St 67, **309**

WIDTH St 67, **310-11**

Wildcards 5-6

WINDOW St 67, **312-13**

World coordinates 235, 236, 312-13

WRITE St 67, **314**

WRITE# St 67, **315**

XOR 35

RADIO SHACK, A Division of Tandy Corporation

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

AUSTRALIA	BELGIUM	FRANCE	U. K.
91 Kurrajong Avenue Mount Druitt. N S W 2770	Rue des Pieds d'Alouette. 39 5140 Naninne (Namur)	BP 147-95022 Cergy Pontoise Cedex	Bilston Road Wednesbury West Midlands WS10 7JN