SPERRY

PERSONAL COMPUTER SYSTEM

BASIC User's Guide

Personal Computer Library
UP-11702 Rev. 2

PERSONAL COMPUTER SYSTEM

BASIC User's Guide

Cursor on/off See LOCATE

Personal Computer Library
UP-11702 Rev. 2

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry representative.

Sperry reserves the right to modify or revise the content of this document. No contractual obligation by Sperry regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry.

PRINTED IN U.S.A.

Preface

This book is a guide for programming in the BASIC* language provided with the SPERRY Personal Computer System (PC), and serves as a reference to the BASIC commands and functions. If you are not familiar with programming in BASIC, you should learn BASIC before using this manual.

> **NOTE:**
>
> This manual provides an overview of BASIC 3.0 for MS-DOS 3.1.

Overview

This book includes the following information:

Chapter 1. Introduction introduces the BASIC programming language.

Chapter 2. Information About BASIC presents some general background information about using the BASIC software.

Chapter 3. BASIC Commands and Statements describes each of the BASIC commands.

Chapter 4. BASIC Functions and Variables describes each of the BASIC intrinsic functions.

Appendix A describes BASIC disk input/output operations.

Appendix B describes BASIC communications interface statements.

Appendix C describes BASIC assembly language provisions.

* This product is named SPERRY BASIC.

Appendix D provides an example of program execution.

Appendix E provides information about converting programs written in other forms of BASIC to SPERRY BASIC.

Appendix F is a summary of error codes and a description of the BASIC error messages the system may display.

Appendix G describes the BASIC mathematical functions.

Appendix H is a summary of the ASCII character codes.

Appendix I describes the typical form of commands, statements, and system variables.

Appendix J provides keyboard scan codes.

An index is located at the back of the book.

Organization of This Book

Before you begin, look briefly through this book. You will notice that the color blue is used extensively. The color is used to make it easier for you to tell what will actually appear on the computer's display screen. The characters you type on the keyboard and the sample screen display responses are printed in blue. Blue is also used to show the format of BASIC commands and highlight examples.

You will also notice that chapter dividers are a different color. A detailed table of contents is printed on each divider page. These divider pages were added to make the book easier to use.

A reader's mail-in card is provided at the back of this book. Please use it to report your suggestions or comments.

Contents

4. BASIC Functions and Variables

Appendixes

A. BASIC Disk I/O

B. Communications

Chapter 1. Introduction

Chapter 1. Introduction

This chapter gives you an overview of the BASIC software supplied with your PC. It tells how to load BASIC and describes some of the commands. A programming language like BASIC is used to create programs to instruct the computer. These programs are stored as program files on your diskettes. This chapter describes how to create simple programs and how to store them and retrieve them from the diskette.

NOTE:

The README.DOC file, if present on the second MS-DOS diskette, contains information such as release notes and errata that is not included in this manual. Be sure to check the file before you use MS-DOS or BASIC. To read the README.DOC file, use your word processor, the EDLIN editor provided with MS-DOS, or the MS-DOS TYPE command.

Loading BASIC

To get BASIC started, the Microsoft Disk Operating System (MS-DOS) must be in operation. If MS-DOS is loaded, the MS-DOS prompt (A>) will be on the left side of the screen. If the MS-DOS prompt is not displayed, refer to the MS-DOS User's Guide, UP-11701.

To load BASIC, type the following command after the MS-DOS prompt:

```
BASIC
```

and press the **Return** or **Enter** key.

This command loads BASIC from the default drive. If you misspelled "BASIC" or BASIC is not present on the default drive, MS-DOS displays the following message:

```
Bad command or file name
```

Insert a diskette containing BASIC.COM in the default drive and try again, or use a drive prefix to tell MS-DOS where to find BASIC. For example, type "B:BASIC" after the MS-DOS prompt to load BASIC from drive B. Remember to press **Return** or **Enter** to execute the command.

The BASIC screen display appears after BASIC is loaded:

```
BASIC n.n VER. n.n

(C) Copyright Microsoft 1983
Created: dd-mmm-yy
xxxxx Bytes free
Ok
```

If BASIC is in operation, you will see the "Ok" prompt. The Ok prompt serves the same function as the MS-DOS prompt. The prompt indicates that BASIC is waiting for you to enter a BASIC command.

NOTE:

This is the simple way to load BASIC. Other options are available to perform special initialization functions (see Chapter 2).

Exiting BASIC

When you are ready to get out of BASIC and return to MS-DOS, type:

```
SYSTEM
```

and press the **Return** or **Enter** key.

This immediately takes you out of BASIC and into MS-DOS. Use the MS-DOS file management system to manage the BASIC programs you create.

BASIC Instructions

A BASIC program is a set of instructions that consists of a keyword followed by parameters describing the operation you want to perform. The keyword specifies an operation.

Keywords in BASIC fall into three categories:

Commands
Statements
Functions

Commands and statements are described in detail in Chapter 3, functions in Chapter 4. For a brief description of each, refer to Appendix I.

Commands

Commands specify the manner in which the program is handled. Commands act on, or control, the execution of the program.

Many BASIC commands sound interesting (COLOR, DRAW, PLAY, etc.). Although their purpose might seem simple, they can be very complicated. Each command is described in detail in Chapter 3.

Example

```
RUN
```

The RUN command tells BASIC to execute the program.

Statements

Statements enable you to access, enter, or manipulate data and specify how the result is produced.

Example

```
10 DATA 1,2,3
20 FOR n= 1 TO 3
30 READ x
40 y=x+1
50 PRINT y
60 NEXT n
70 END
```

In this example, the DATA and READ statements make the data available to the program. The FOR. . .NEXT statement and the computation in line 40 manipulate the data. The PRINT statement specifies how the result is output.

Functions

Functions are predefined mathematical operations, string manipulation operations, and operations you can define. A function does not stand alone; it is part of a statement.

Example

```
PRINT INT (345.67)
PRINT LEN ("cloudy skies")
```

The PRINT statement is one of the simpler statements. Some statements are more complex, especially those dealing with graphics and sound.

File Handling Commands

Some commands in BASIC are similar to commands in MS-DOS. They are used to manage the files on your diskettes. These commands manage files created in BASIC or files that already exist on your diskette (created in MS-DOS).

The command you use in BASIC may be different from the command you use in MS-DOS, even though it has the same purpose. For example, the BASIC command, FILES, does the same thing as the MS- DOS command, DIR. Both ask the system to display (on the screen) a list of the files currently stored on the diskette.

In MS-DOS, you enter the DIR command next to the "A>" prompt. In BASIC, you enter the FILES command (in either uppercase or lowercase letters) in the first column on the left side of the screen below the "Ok" prompt. You don't have to exit BASIC to do this, even though you may be displaying MS-DOS files. Just type:

```
FILES
```

and press the **Return** or **Enter** key.

The system displays the contents of the default drive. The display looks something like this:

```
Ok
files
A:\
COMMAND .COM  ANSI     .SYS  FORMAT   .COM  CHKDSK   .COM
SYS     .COM  DISKCOPY .COM  DISKCOMP .COM  COMP     .COM
EDLIN   .COM  MODE     .COM  FDISK    .COM  BACKUP   .COM
RESTORE .COM  PRINT    .COM  RECOVER  .COM  ASSIGN   .COM
TREE    .COM  GRAPHICS .COM  SORT     .EXE  FIND     .EXE
MORE    .COM  EXE2BIN  .EXE  LINK     .EXE  DEBUG    .COM
ROMVER  .COM  CLS2     .COM  CLS3     .COM  BASIC    .EXE
DEMO    .BAS  CHAR     .BAS  COLORI   .BAS  COLORII  .BAS
BUTTERF .BAS  CIRLIN   .BAS  BUBBLE   .BAS  CHARACT  .BAS
    6144 Bytes free
Ok
```

BASIC commands that are similar to MS-DOS commands are listed in the following table.

BASIC Command	DOS Command	Function
FILES	DIR	Used to display the names of the files that are stored.
NAME	RENAME	Used to change the name of a file.
KILL	DEL	Used to delete a file.
DATE$	DATE	Used to ask the system to display the current date.
TIME$	TIME	Used to ask the system to display the current time.

An example of a BASIC program that uses file handling commands is included in Appendix D.

Writing BASIC Programs

A BASIC program is a series of steps. Each step is a command.

This section, an overview of the steps involved in writing a BASIC program, is designed to help beginning programmers.

To write a BASIC program, follow these procedures:

First, you need a line number.

Commands are sent to the system in sequence. Each step in the sequence is entered on a new line. The system determines the next step in the sequence by looking for the next line number.

The order in which you enter the line numbers doesn't really matter. BASIC sorts them later. The system reads and acts upon the lowest numbered line and continues to the highest numbered line. Then it stops and returns to the beginning BASIC screen.

Generally, you enter one BASIC command on each line. For example, the PRINT command tells the system to print (on the screen) the words or numbers that follow PRINT. It's a good idea to number your lines by 10's or 100's because you might forget something and want to add new lines later.

Enter a program like this by pressing the **Return** or **Enter** key after each line:

```
10 PRINT 1
20 PRINT 2
30 PRINT 4
```

This little program will print the numbers listed, one number on each line.

Later, if you decide you want to print the number 3 after the number 2, add another line, like this:

```
25 PRINT 3
```

The computer stores it in the proper line number sequence:

```
10 PRINT 1
20 PRINT 2
25 PRINT 3
30 PRINT 4
```

When you run the program, the system acts on each line in sequence, printing 1 on the first line, 2 on the second line, and so on.

NOTE:

Several common words have special meaning in BASIC. They are listed and defined below:

RUN Tells the system to go through the sequence of lines and perform the commanded operations.

LIST Tells the system to list on the screen all the lines entered so far.

LOAD Tells the system to load a program (it must already be stored on the disk) so it can be run.

SAVE Tells the system to save the program by writing its contents on the disk.

Second, you have to enter a BASIC command.

After you enter the line number, you must enter a BASIC command the system recognizes (Chapter 3 includes all of the commands). Type the BASIC command next to the line number (with a space after the line number) like this:

```
10 PRINT "aaabbbccc123"
```

Third, enter any data you want that command to act on.

In the previous example, line number 10 is followed by a space, the PRINT command, and another space. The system recognizes this as a command to print whatever follows. In this case, the system prints the words within quotation marks.

When you run the program, the system acts on the command in line 10 after it executes any line numbered lower than 10 and before any line numbered higher than 10 (if any).

Example

```
10 PRINT "aaabbbccc123"
```

Notice that the phrase to be printed is enclosed in quotation marks. Any time you ask the system to print letters (or a combination of letters and numbers), you must enclose them in quotation marks. The quotation marks are not printed. Only the characters inside the quotation marks are printed.

If you are printing only numbers, you do not need to enclose them in quotation marks.

The preceding example is a complete program. There is no minimum number of lines or commands in a BASIC program. There is no maximum either. The only limit is the amount of memory storage space your personal computer has. The PC has enough room for programs with thousands of lines.

After you enter a line with a command, you can tell the system to run that program. In the preceding example, it would print the following message:

```
aaabbbccc123
```

Writing Your Own Programs

Try writing a simple program.

Remember the three essential steps:

1. Type a line number (and leave a space).
2. Type a BASIC command (leave another space).
3. Type any data you want that command to act on.

The PRINT command is a good command to practice. It is used in almost every program. Here's how you program the system to print a message on the screen using PRINT command:

```
10 PRINT "Program number 1"
20 PRINT
30 PRINT "This is my first program."
```

Try this mini-program. (Remember to press the **Return** or **Enter** key at the end of each line.)

If you tell the system to run the program (type RUN and press the **Return** or **Enter** key), the following message should appear on the screen:

```
Program number 1

This is my first program.
```

In line 20, the PRINT command is not followed by data. This results in a blank line. The system does exactly what you tell it to do. If you want spaces between words or between lines, you have to tell the system to do it.

Once your program is stored, all you have to do is type the word RUN and press the **Return** or **Enter** key. The message is printed on the screen again. Your program is automatically stored in the computer's memory and stays there until you begin a new program by typing the word NEW (or until you turn the system off).

NOTE:

When a program has problems, programmers often say it has a bug. This usually means some simple problem was overlooked while you were writing the program. If your 3-line print program has a bug in it and won't print the message on the screen, look at what you entered. Make sure you entered a line number and left the right number of spaces. Be sure you spelled PRINT correctly and used quotation marks before and after the message to be printed on the screen. Don't forget to press the **Return** or **Enter** key after typing each line.

Saving Your Programs

Your program is now stored in temporary memory inside the personal computer (PC). If you want to store it on a disk, you must tell the PC to do so.

If the "Ok" prompt is on the left of the screen, type SAVE and leave a space. Then type quotation marks, followed by the name you want to call your program. The name can be eight characters long.

The program will be stored on the disk by that name, followed by a period (.) and the extension BAS (for BASIC).

Try saving the 3-line PRINT program you just wrote. Let's say you want to name the program my1. Type the word SAVE, followed by a space. Then, with no space, type "my1.

Here is how it should appear on the screen:

```
SAVE "my1
```

The SAVE command requires quotation marks only in front of the program name; you don't need quotation marks after the name.

When you press the **Return** or **Enter** key, the program is saved on the disk in the drive from which you loaded BASIC (the default drive). The program is stored as MY1.BAS. (When you omit the extension, BASIC saves programs with a .BAS extension.)

If you want to store a program on a disk in another drive, you must specify the name of the drive in the SAVE command. For instance, if the default drive is drive A and you want to store a program on the disk in drive B, you must type the letter b and a colon (b:) in front of your filename, as follows:

```
SAVE "b:my1
```

In the above example, the program will be stored in drive B.

> **NOTE:**
>
> To load and run the same program later, you must specify the disk drive again.

In this example, to be sure the program was saved on your disk, type:

```
FILES "B:"
```

and press **Return** or **Enter**. The names of the files on drive B scroll up on the screen.

To list the names of the files on the default drive, enter FILES without a drive specification, for example:

```
FILES
```

and press **Return** or **Enter**.

Note that you can also use the MS-DOS command to list the contents of a disk, but you have to return to MS-DOS first by entering the SYSTEM command (see the MS-DOS User's Guide).

If your program was saved in the MS-DOS system, it will be listed in the specified directory as follows:

```
MY1.BAS  xx  6-21-83  10.23a
```

(Where xx represents a number that indicates how much disk space the program took).

The date and time the program was stored is useful information; you can use it to keep track of several versions of the same program by noting the date and time each was stored.

Running a BASIC Program

Once you have written a BASIC program and stored it on a diskette, you can run it any time you want.

If you are in BASIC, type the word LOAD, leave a space, and type quotation marks and the name of the program you want to load.

```
Ok
LOAD "my1
```

The system will find the program my1 on the disk and load it into memory. Enter the word RUN and press the **Return** or **Enter** key. The program will run.

Remember, to load and run a program that has been saved on a disk in a drive other than the default drive, you must specify that drive in the LOAD command, as follows:

```
Ok
LOAD "b:my1
```

You don't have to use the SAVE command to store the program again. Even if you use the NEW command to clear the computer's memory, the original copy of my1 that you saved on the disk is still stored there. You can load a program as often as you want.

NOTE:

Whenever BASIC is loaded, a row of words is displayed along the bottom of the screen. These words are reminders that the function keys **F1** through **F10** are programmed and active. Each key can be programmed to perform a complicated function when you press it. In BASIC, several function keys have already been programmed. To run a program, you usually have to type the word RUN and press the **Return** or **Enter** key. But function key **F2** runs the program that is currently active in memory. For descriptions of the preprogrammed function keys, refer to Table 2-3.

Learning to Program

Now that you know how to write and save BASIC programs, try to write some of your own. When you write programs, it's a good idea to begin with a plan of what you want the system to do.

Many books that teach the principles of programming in BASIC contain sample programs you can copy and store on a diskette. Copying programs is a good way to learn to program. Carefully copy the program and make sure it runs; then, store it on a disk using the SAVE command. Often, you can tailor an existing program to fit your needs.

You can test your practice programs by running each program at various stages of development. As you add a feature, try running it before you add more. If your modifications don't work, you can always cancel them by typing NEW and loading the original program again (it will still be stored on the diskette).

Running Other Programs

Your SPERRY BASIC software can load and run the programs you write, as well as many other programs written in a form of BASIC compatible with SPERRY BASIC. Any program that includes the .BAS extension as part of its name should run on your SPERRY PC, but you may have to modify it to account for minor differences in command structure. (Refer to Appendix E.)

To see if a program will run, insert the diskette containing the program into the default drive. Use the LOAD command to load the program into active memory. Then use the RUN command.

If the program won't run properly, use the LIST command to check the lines that have errors. Sometimes you can fix a program yourself by changing a few format problems and learn a lot about programming at the same time. (Be sure to make a backup copy of the program before you try to modify it.)

Some application programs require BASIC. If you attempt to run such a program without BASIC on the default drive, the following message will appear on the screen:

```
Bad command or filename
```

This means the operating system cannot find BASIC. To run the program, use the MS-DOS COPY command to copy your BASIC files (BASIC.COM, BASICA.COM, and BASICA.EXE) onto the disk in the default drive.

If there isn't enough room on the disk, copy only BASIC.COM and BASICA.COM. By executing BASIC.COM or BASICA.COM, BASICA.EXE is invoked. If BASICA.EXE is not found in the current directory or established paths, you will see the following prompt:

```
Type in path name containing BASICA.EXE
and (Y/N) to indicate if any disk needs
to be replaced
***Do not replace disk yet***
Format: <path name>[,Y or N]
```

Enter the path name (explained in Chapter 2) containing BASICA.EXE and "Y" or "N" to indicate whether or not you will be replacing any disk.

If you enter the path name and N (no), BASICA.EXE will be read into memory.

If you enter the path name and Y (yes), drive A: is assumed and BASIC will display the following prompt:

```
Insert the disk containing BASICA.EXE
in drive A: and strike any key when ready
```

After replacing your diskette, BASICA.EXE will be read into memory and the following prompt will appear:

```
Replace the disk in drive A: and
strike any key when ready
```

You can now reinsert your application disk and run the program.

Sample Programs

Some sample programs are included on your BASIC diskette. You can load and run these programs to test the capabilities of your PC. Use the DIR command to get a listing of the files on the diskette that include the .BAS extension and load and run them in the usual way.

Some sample programs demonstrate color graphics. If you are not using a color display monitor with your system, these programs will not run properly.

Chapter 2. Information About BASIC

Figures

Tables

Chapter 2. Information About BASIC

BASIC Format

Format Notation

Each command, statement, and function in SPERRY BASIC is described using the following notation conventions.

Words in capital letters are keywords. You may enter keywords in uppercase or lowercase. BASIC converts them to uppercase unless they are part of a quoted string or remark.

Examples

```
PRINT
FOR. . .NEXT
```

Lowercase items between greater than and less than signs must be supplied by you.

Examples

```
<duration>
<array name>
```

Items enclosed in square brackets are optional.

Examples

```
[<record number>]
[STEP z]
```

You must enter all punctuation marks (commas, parentheses, quotation marks, etc.), except the greater than and less than signs (< >) and square brackets ([ ]) that are used for notation. In addition, command keywords must be followed by one or more spaces. Note that spaces are often used in this manual to make examples and the representation of command syntax clearer. Other than the required space following a command keyword, spaces have no special significance except to make your programs more readable.

Examples

```
CHDIR "<path>"
GET (<x1>,<y1>)-(<x2>,<y2>),<array name>
```

Ellipses (...) indicate that an item may be repeated.

Line Format

A program line begins with a line number and ends with a carriage return (caused by pressing the **Return** or **Enter** key). Each program line may contain up to 255 characters.

```
nnnn BASIC statement [:BASIC statement...]
```

Line Numbers

Line numbers specify the order in which program lines are stored in memory. They are also used as references for branching and editing. Line numbers can be from one to five digits (0 to 65529).

You may use a period to refer to the current line number in EDIT, LIST, AUTO, and DELETE commands. For example, if the current line number is 640 and you want to delete it, enter:

```
DELETE .
```
or
```
DELETE 640
```

BASIC Statements

BASIC statements consist of keywords followed by variable names or values. A space must separate the keyword from the rest of the statement.

Multiple Statements

You may enter multiple statements on a line, but you must separate each statement on the line with a colon. For example:

```
10 FOR x=1 TO 10: PRINT x: NEXT x
```

To extend a logical program line beyond the physical screen, just continue typing. The cursor moves to the next line on the screen.

Programming

Initialization

Before you begin programming, you must initialize (load) the system. The format used to load BASIC from the MS-DOS system is:

```
BASIC [<filename>] [< <stdin> >] [<stdout>]
BASIC [<filename>] [< <stdin>] [> <stdout>]
[/M:<highest memory location>][,<max blocksize>]
[/C:<combuffer>] [/D] [/P:[<load address>]] [/G]
```

Loading BASIC without specifying a <filename> is explained in Chapter 1. If you use a valid filename, BASIC proceeds as if a RUN statement were entered. If you don't include a filename extension, BASIC assumes .BAS by default.

BASIC allows you to redirect input and output. The <stdin> and <stdout> options stand for "standard input"and "standard output." By default, BASIC reads its input from the keyboard, and writes its output to the display monitor. The <stdin> and <stdout> options are the names of files or devices. When you specify <stdin>, BASIC reads its input from the file instead of the keyboard. Similarly, when you specify <stdout>, BASIC writes its output to the file instead of the display monitor.

For example,

```
BASIC BFILE < INFLE > OUTFLE
```

tells the system to load the BASIC interpreter and run the BASIC program BFILE.BAS. The left angle bracket (<) tells BASIC to read its input from the file INFLE, and the right angle bracket (>) tells BASIC to write its output to the file OUTFLE.

When used, the <stdin> and <stdout> options must precede all other options.

/M:<highest memory location> sets the highest memory location to be used by BASIC. It may be expressed in decimal, octal (preceded by &O), or hexadecimal (preceded by &H) numbers. /M:<max blocksize> allows you to load programs above the BASIC workspace. <max blocksize> must be in byte multiples of 16.

/C:<combuffer> sets the size of the buffer used to receive data through communication files.

The /D option allows double precision calculations using the ATN, COS, EXP, LOG, SIN, SQR, and TAN functions. If /D is omitted, calculations are performed in single precision and the space is freed for program use.

/P:[<load address>] allows the BASIC loading address to be specified. If <load address> is omitted, BASIC is loaded at the lowest available memory location. <load address> must be in paragraphs (byte multiples of 16). There are very few circumstances that require the use of the /P: switch. It is recommended that this switch be omitted unless it is specifically called for.

/G specifies that 25 screen lines are used for scrolling. After executing a KEY OFF statement and reaching the end of line 24, character output continues on line 25. If /G is not specified, 24 screen lines are used for scrolling. The bottom line of the screen is not scrolled. The LOCATE statement can set the cursor on line 25.

Examples

A: BASIC Payroll.bas	Using all memory, load and execute Payroll.bas.
A: BASIC Invent	Execute Invent.bas.
A: BASIC /M:32768	Use first 32K bytes of memory.
A: BASIC Datack /M:&H9000	Use first 36K bytes of memory and execute Datack.bas.

Command and Program Modes

When BASIC is loaded, "Ok" is displayed on the screen. "Ok" means BASIC is ready to accept commands or statements. BASIC may be used in one of two modes: command mode or program mode.

Command Mode

If you enter a command or statement in command mode, it is executed immediately. You do not need to precede program lines with line numbers. Just type the command and press **Return** or **Enter**.

Example

```
Ok
PRINT 10+02
 12
Ok
```

Although results of arithmetic and logical operations are displayed immediately, the instructions are removed from memory after execution. The command mode is useful for debugging and for quick computations.

Program Mode

In program mode, you precede statements with line numbers. Statements are stored in memory to form a complete program. To execute the program, use the RUN command.

Example

```
10 PRINT 10+02
RUN
 12
Ok
```

Use program mode when you want to perform a series of repeated or complex processes. For example, you can write a payroll program once and use it repeatedly. (Chapter 1 describes how to store a program.)

How to Enter a Program

To enter a program in BASIC:

1. Enter a NEW command.

   The BASIC program area is cleared so you can enter a new program.

2. Enter an AUTO command to automatically assign a line number to each statement you enter. (You can clear this function by pressing the **Ctrl** and **Break** keys simultaneously.)

   This command is optional. If you don't use it, you have to type each line number.

3. Enter program statements sequentially. To correct a typing error:

   a. Erase the preceding character or characters by pressing the **Back Space** key.

   b. Erase the whole line being entered by pressing the **Esc** key.

4. To correct a line you have already entered, refer to the next section.

5. After you have entered all the necessary statements, press **Ctrl Break** to clear the AUTO command function.

6. Enter a LIST command to display the program you just entered and check each statement for typing errors.
7. Enter a RUN command to execute the program.
8. To store the program in a disk file, use the SAVE command.

Try writing a program to find the real root of a quadratic equation.

```
Ok
NEW
Ok
AUTO
10 INPUT "COEFFICIENT A=";A
20 INPUT "COEFFICIENT B=";B
30 INPUT "COEFFICIENT C=";C
40 D=B ^2-4*A*C
50 IF D<0 GOTO 110
60 x1=(-B+SQR(D))/(2*A)
70 x2=(-B-SQR(D))/(2*A)
80 PRINT "x1=";x1
90 PRINT "x2=";x2
100 STOP
110 PRINT "No real root"
120 END
```

Press the **Ctrl** and **Break** keys.

```
Ok
RUN
COEFFICIENT A=?1
COEFFICIENT B=?2
COEFFICIENT C=?1
x1=-1
x2=-1
Break in 100
Ok
```

Correcting the Current Program

You can correct a BASIC program in one of two ways: line-by-line or character-by-character.

Line-By-Line Program Corrections

To correct a program line-by-line:

1. Check the contents of the program.

   Use the LIST command to check the contents of the program. (If the program is still on disk file, use a LOAD command before using the LIST command.)

   If you need to interrupt the listing, press **Ctrl Num Lock**. To restart the listing, press any key.

   To cancel the LIST command, press **Ctrl** and **Break** simultaneously. The system displays the program in standard format.

2. Replace a program line.

   You can replace (change) a program line by entering the old line number and a new command.

3. Add or insert a program line.

   You can add or insert a new line by assigning a new line number.

   For example, to add a line to a program that ends with line 100, use line number 110. To insert a line between lines 70 and 80, use line number 75. To replace line 50, use line number 50.

4. Delete a program line.

   To delete a program line, use the DELETE command and the line number. To delete lines within a certain range, use DELETE and the range of line numbers.

Examples

DELETE 600	Delete only line 600.
DELETE 300-400	Delete all lines from 300 to 400 (inclusive).
DELETE-500	Delete all lines up to line 500 (inclusive).
DELETE 500-	Delete all lines from line 500 (inclusive).
DELETE .	Delete the current line.

5. Check the corrected program.

Use the LIST command with the range of line numbers to check the corrected program.

Examples

LIST	Display the whole program.
LIST 300-400	Display all lines from 300 to 400 (inclusive).
LIST-500	Display all lines up to line 500 (inclusive).
LIST 600-	Display all lines from line 600 (inclusive).

Character-by-Character Program Corrections

To perform character-by-character corrections on an existing program, use the cursor control keys and editing keys to insert, add, or delete characters displayed on the screen. (Refer to the "Keyboard" section for more information on cursor control and editing keys.)

To correct a program character-by-character:

1. Use a LIST or EDIT command to display the program line you want to correct.

   If you use an EDIT command, only the line you specify is displayed. The cursor is positioned at the beginning of the line.

2. To move the cursor into position for correction, use the cursor control keys.
3. Use the editing keys to correct characters.
4. Once you have corrected a line, press the **Return** or **Enter** key. This replaces the old line with the corrected line. (If you do not press **Return** or **Enter** after you make a correction, the old line remains unchanged in memory.)

BASIC Language Components

Character Set

Alphabetic characters in BASIC are all uppercase and lowercase letters of the alphabet. Numeric characters are 0 through 9. In addition, the following special characters are recognized by BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent sign
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less-than sign
>	Greater-than sign
\	Backslash or integer division symbol
_	Underscore
"	Double quotation mark

Reserved Words

Reserved words are keywords that have special meaning in BASIC. They include commands, statements, operator names, and function names. You cannot use reserved words as variables, but you use them as a part of a variable name.

Always separate reserved words from other syntax units with commas or spaces (delimiters).

Reserved words in BASIC are:

ABS	CVI	FIELD	LEN
AND	CVS	FILES	LET
ASC	DATA	FIX	LINE
ATN	DATE$	FNxxxxxxxx	LIST
AUTO	DEF	FOF	LLIST
BEEP	DEFDBL	FRE	LOAD
BLOAD	DEFINT	GET	LOC
BSAVE	DEFSNG	GOSUB	LOCATE
CALL	DEFSTR	GOTO	LOF
CDBL	DELETE	HEXS	LOG
CHAIN	DIM	IF	LPOS
CHP$	DRAW	IMP	LPRINT
CINT	EDIT	INKEY$	LSET
CIRCLE	ELSE	INP	MERGE
CHDIR	END	INPUT	MID$
CLEAR	ENVIRON	INPUT$	MKD$
CLOSE	ENVIRON$	INPUT$	MKDIR
CLS	EOF	INSTR	MKI$
COLOR	EOV	INT	MKS$
COM	ERASE	INTER$	MOD
COMMON	ERDEV	IOCTL	MOTOR
CONT	ERDEV$	IOCTL	NAME
COS	ERL	KEY	NEW
CSNG	ERR	KEY$	NEXT
CSRLIN	ERROR	KILL	NOT
CVD	EXP	LEFT$	OCT$

OFF	PSET	SHELL	TROFF
ON	PUT	SIN	TRON
OPEN	RANDOMIZE	SOUND	USING
OPTION	READ	SPACE$	USR
OR	REM	SPC (	VAL
PUT	RENUM	SQR	VIEW
PAINT	RESET	STEP	VARPTR
PALETTE	RESTORE	STOP	VARPTR$
PEEK	RESUME	STR$	WAIT
PEN	RETURN	STRING$	WINDOW
PLAY	RIGHT$	SWAP	WEND
PMAP	RMDIR	SYSTEM	WHILE
POINT	RND	TAB (	WIKTH
POKE	RSET	TAN	WRITE
POS	RUN	THEN	WRITE#
PRESET	SAVE	TIME	XOR
PRINT	SCREEN	TIMER	
PRINT#	SGN	TO	

Constants

Constants are the actual values BASIC uses during program execution. There are two types of constants: string and numeric.

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

Examples

```
"HELLO"
"$25,000,00"
"Number of Employees"
```

A numeric constant is a positive or negative number. Numeric constants cannot contain commas. The five types of numeric constants are:

1. Integer constants — Whole numbers between −32768 and +32767. (Integer constants do not have decimal points.)

2. Fixed point constants — Positive or negative real numbers (numbers that contain decimal points).

3. Floating point constants — Positive or negative numbers represented in exponential form (similar to scientific notation).

   A floating point constant consists of an integer of fixed point number (the mantissa) followed by the letter E and another integer (the exponent). Integers may be positive or negative. The allowable range for floating point constants is $10^{(-38)}$ to $10^{(+38)}$.

   Examples

   235.988E−7 = .0000235988
   2359E6 = 2359000000

   (Double precision floating point constants use the letter D instead of E.)

4. Hex constants

Digits of the base 16 hexadecimal numbering system. Hex constants have the prefix &H followed by up to 4 digits (0–9 and A–F inclusive).

Examples

```
&H76
&H32F
```

5. Octal constants

Digits in the octal numbering system. Octal constants have the prefix &O or & followed by up to 6 digits (0–7 inclusive).

Examples

```
&O347
&1234
```


Numeric constants in BASIC may be single precision or double precision numbers. Double precision numbers are stored within 16 digits of precision and printed with up to 16 digits.

A single precision number is any numeric constant that has:

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

Examples

```
46.8
-1.09E-06
3489.0
22.5!
```

A double precision number is any numeric constant that has:

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

Examples

345692811
-1.09432D-06
3489.0#
7654321.1234

Variables

Variables are names assigned to storage locations that contain values used in a BASIC program. You may assign the value of a variable (as a constant) or it may be the result of calculations in the program. Before a variable is assigned a value, its assumed value is zero.

There are two types of variables: string variables and numeric variables. A string variable always has character string value, and a numeric variable always has a numeric value. For example, suppose that string variable S$ contains the characters "123". Internally, each character in the string is represented by the ASCII code for that character. In comparison, numeric variables are stored in internal, numeric format.

You cannot perform arithmetic on string variables directly. The expression A$ = B$ + C$ puts the contents of B$ and C$ together and stores the combined string in variable A$. This process is called concatenation. If B$ contains the characters "123" and C$ contains "456", then the expression A$ = B$ + C$ causes the combined string "123456" to be assigned to variable A$.

If a string variable contains valid numeric characters, you can use the VAL function to convert character string data to numeric data. The expression A = VAL(B$) + VAL(C$) converts the "123" in B$ and "456" in C$ to numeric values. When these values are added together, the sum (579) is assigned to numeric variable A.

Similarly, you can use the STR$ function to convert numeric values to string data. The expression A$ = STR$(123) + STR$(456) concatenates the character representation of the numbers 123 and 456, and assigns the result to A$.

Variable Names and Declaration Characters

Variable names in BASIC may be any length, but only the first 40 characters are significant. Letters, numbers, and the decimal points are allowed, but the first character must be a letter. Type declaration characters ($, %, !, #) are allowed at the end of the variable name.

You cannot use a reserved word as a variable name, but reserved words may be included within the variable name. (If a variable name begins with FN, the system treats it as a call to a user-defined function.) All BASIC commands, statements, function names, and operator names are reserved words.

The last character of a variable name determines the type of constant that can be stored. For example, a string variable name must end with a dollar sign so that characters may be stored.

Example

```
A$ = "SALES REPORT"
```

The last character of a numeric variable name depends on whether you want to store an integer, single precision, or double precision number. The characters used to declare numeric variables (declaration characters) are:

%	declares an integer variable
!	declares a single precision variable
#	declares a double precision variable

Any variable name that does not end with $, %, !, or # is treated as a single precision numeric variable.

Examples

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

Array Variables

An array is a group (table) of values with a common variable name. For instance, an array name may represent rows or columns of data arranged in storage. Each individual value in an array is called an element. Each element is distinguished by a subscript. A subscript may be an integer or integer expression.

An array has as many subscripts as it has dimensions. For example, V(10) refers to a value in a 1-dimensional array. T(1,4) refers to a value in a 2-dimensional array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

Space Requirements

Variables:

Integer	2 bytes
Single precision	4 bytes
Double precision	8 bytes

Arrays:

Integer	2 bytes per element
Single precision	4 bytes per element
Double precision	8 bytes per element

Strings:

3 bytes overhead plus the present contents of the string

Type Conversion

When necessary, BASIC converts numeric constants from one type to another. Note the following rules and examples:

1. If you try to assign one type of a numeric constant to a different type of numeric variable, the number stored is the type declared in the variable name. You cannot assign string constants (characters) to numeric variables (or vice versa).

   Example

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

2. Whenever the system evaluates an expression, all operands are converted to the same degree of precision as that of the most precise operand. Also, the result of an arithmetic operation has the same degree of precision.

   Example

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

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

Example

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

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

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range −32768 to +32767 or an "Overflow" error occurs.
4. When a floating point value is converted to an integer, the fractional portion is rounded.

Example

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

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number are valid. This is because only seven digits of accuracy were supplied with the single precision value.

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

Example

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

Expressions and Operators

An expression is a string or numeric constant, a variable, or a combination of constants and variables (with operators) to produce a single value.

Operators perform mathematical or logical operations on values.

There are four categories of operators:

Arithmetic
Relational
Logical
Functional

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	x^y
-	Negation	-x
*	Multiplication	x*y
/	Floating Point Division	x/y
\	Integer Division	x\y
MOD	Modulo Arithmetic	x MOD y
+	Addition	x+y
-	Subtraction	x-y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

ALGEBRAIC EXPRESSION	BASIC EXPRESSION
$x + 2y$	x + 2*y
$x - \frac{y}{z}$	x - y/z
$\frac{xy}{z}$	x*y/z
$\frac{x + y}{z}$	(x + y)/z
$(x^2)^y$	(x ∧ 2)∧y
x^{y^z}	x ∧ (y ∧ z)
x(-y)	x*(-y)

(Two consecutive operators must be separated by parentheses.)

Integer Division and Modulo Arithmetic

Use the backslash (\ ) to perform integer division. The operands are rounded to integers (in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

Example

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

Use the operator MOD to perform modulo arithmetic. It gives the integer a value that is the remainder of an integer division.

Example

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

Overflow and Division by Zero

If overflow occurs, an error message is displayed on the screen. The result of the operation is machine infinity. Program execution continues.

Division by zero produces the following results:

1. An error message is displayed on the screen.
2. The result of the division is machine infinity with the sign of the numerator.
3. Program execution continues.

An attempt to raise zero to a negative power produces the following results:

1. An error message is displayed on the screen.
2. The result of this exponentiation is positive machine infinity.
3. Program execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either true (–1) or false (0). You can use this result to make a decision regarding program flow.

Operator	Relation Tested	Expression
=	Equality	x=y
<>	Inequality	x<>5
<	Less than	x<y
>	Greater than	x>y
<=	Less than or equal to	x>=y
>=	Greater than or equal to	x>=y

The equal sign is also used to assign a value to a variable.

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

```
x+y < (T-1)/z
```

is true if the value of x plus y is less than the value of T–1 divided by z.

Examples:

```
IF SIN(X) <0 GOTO 1000
IF I MOD J < > 0 THEN K=K+1
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a result which is either true (not zero) or false (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined in bitwise fashion.

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision.

Example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators convert their operands to 16-bit, signed twos that complement integers in the range –32768 to + 32767. (If the operands are not in this range, an error results.) If you supply both operands as 0 or 1, logical operators return 0 or –1. The given operation is performed on these integers in bitwise fashion (each bit of the result is determined by the corresponding bits in the two operands) as shown in the following chart.

NOT	X	NOT X
	1	0
	0	1

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

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

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

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

EQV	X	Y	X EQV Y
	1	1	1
	1	0	0
	0	1	0
	0	0	1

It is possible to use logical operators to test bytes for a particular bit pattern. For instance, you may use the AND operator to mask all but one bit of a status byte at a machine I/O port. You may use the OR operator to merge two bytes and create a given binary value.

Examples

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1
	The bit complement of 16 zeros is 16 ones, which is the two's complement representation of -1
NOT x = -(x+1)	The two's complement of any integer is the complement of the bit value plus one

Functional Operators

A functional operator is used to call a predetermined operation that is to be performed on an operand. BASIC has functions that reside in the system, such as SQR (square root) or SIN (sine). BASIC's intrinsic functions are described in Chapter 4.

BASIC also allows user-defined functions written by the programmer.

String Operations

You may concatenate strings using +.

Example

```
10 A$="File" : B$="name"
20 PRINT A$+B$
30 PRINT "New "+A$+B$
RUN
Filename
New Filename
```

Strings may be compared by using the same relational operators used with numbers:

= <> < > <= >=

You can make string comparisons by comparing the ASCII codes of each character in a string. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher one. If, during string comparison, the end of a string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples

```
"AA" < "AB"
"filename" = "filename"
"X&" > "X#"
"CL" > "cl"
"KG" > "Kg"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"        (where B$ = "8/12/78")
```

You can use string comparisons to test string values or to alphabetize strings. Enclose string constants used in comparison expressions in quotation marks.

Input and Output Operations

Input and output (I/O) operations in a data processing system include entering data, accessing data, and obtaining results. I/O features include:

Files
Directories
Paths
The screen
Advanced features (clock and sound)
The printer

Files

A file is a collection of data stored in a device (other than the memory). Files are usually stored on a diskette or a fixed disk and are accessed by opening them (calling them by name). After you have opened a file, you can perform I/O activities.

There are two types of BASIC files: program files and disk data files.

Program files are the programs you write. Disk data files (sequential and random access) are described in Appendix A.

Naming and Accessing Files and Devices

To name or access any file, use the following format:

[<device name>:][<filename>[.<extension>]]

The device name specifies the device used for an I/O operation, and the filename specifies a file on that device. The filename extension is optional. When you specify a device, you must use the colon, even if you don't specify a file (or another option).

The device name, filename, and extension are known as the filespec.

Examples

```
b:apples.231
SCRN:
```

Device Names

Device names refer to the I/O equipment you are using. Devices include the keyboard, the screen, disk drives, printers, and communications adapters.

Each device must be assigned a name. The name may have up to four characters. Allowable device names are listed in Table 2-1.

You must follow the diskette drive name with a filename. You can, however, omit the device name if you are using the drive that contains the file you have specified.

Table 2-1. Valid Device Names

Device Name	Type of I/O Device	Input	Output
KYBD:	Keyboard	o	x
SCRN:	Screen	x	o
LPT1:	Printer	x	o
LPT2:	Printer	x	o
LPT3:	Reserved	—	—
COM1:	RS-232-C unit number 0	o	o
COM2:	RS-232-C unit number 1	o	o
COM3:	Reserved	—	—
COM4:	Reserved	—	—
A:	Diskette drive A	o	o
B:	Diskette drive B	o	o
C:	First fixed disk or V disk	o	o
D:	Second fixed disk or V disk	o	o

o = permitted
x = not permitted

Filenames

You must use a filename to identify a file. The filename format is:

<filename>[.<extension>]

Examples

payroll.2rl
payroll.bas
(ledger!.121

To name a file, follow the rules provided by MS-DOS:

1. A filename may have up to eight characters. An extension may have up to three characters.
2. If you use an extension longer than three characters, only the first three characters are valid. The remaining characters are truncated.
3. If a filename has more than eight characters, the first eight characters become the filename.
4. If the filename has more than eight characters and an extension, the first eight characters become the filename and the first three characters after the period become the extension.
5. If you omit the extension, .BAS is automatically added to the filename (unless you have used a KILL, NAME, or OPEN statement).
6. You may use the following characters for filenames and extensions:

   A through Z (uppercase or lowercase)
   0 through 9
   symbols ( ) { } @ # $ % < & ! - _ ' / ~ ¦

   The symbols <, >, and \ (supported in BASIC 1.0) are not valid in this version of BASIC.

Directories

A new method of file organization has been developed for BASIC. Previously, there was one directory with as many individual files as the diskette could hold. This version enables you to organize files in a directory hierarchy called a tree-structured directory.

An overview of directory concepts is included here. For more information on tree-structured directories, refer to the MS-DOS User's Guide.

Root Directories

The root directory (system directory) is at the top of the tree. It contains filenames and directory names. These directories may contain more directories, which contain more directories, and so on.

A directory is a file that contains multiple documents. A root directory may contain many files and directories. The number of directory levels depends upon available diskette or fixed disk space.

Subdirectories

Directories within the root directory are called subdirectories.

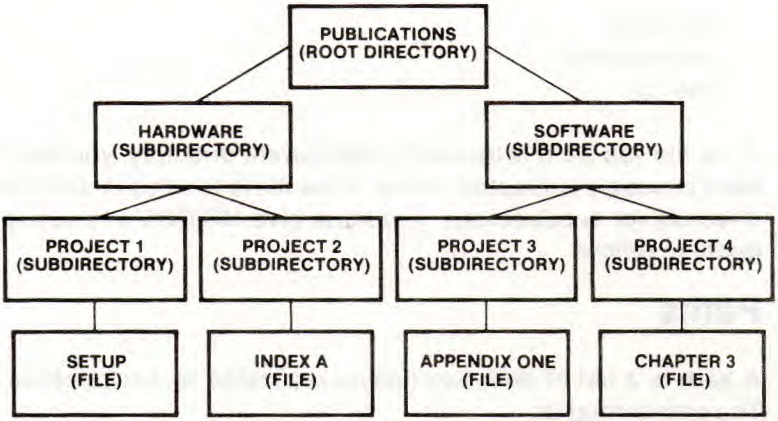
Directories are created using an MKDIR command and deleted using an RMDIR command. Refer to Chapter 3 for more information on these commands.

Files

Files are locations that contain data. A subdirectory or root directory may contain multiple files.

Tree-Structured Directories

The tree-structured directory is organized as follows:



Naming and Accessing Directories

Directory Names

Directory and subdirectory names use the same format as filenames. Up to eight characters and an optional 3-character extension are allowed. A directory may contain filenames that appear in other directories. Duplicate directory names, however, will cause an error.

Current Directory

The current directory operates much like a default diskette drive. After you inform MS-DOS of the directory you want to access, MS-DOS remembers this location. To change directories, use the CHDIR command (Chapter 3). The root directory is the current directory unless you specify another directory.

Locating a File

In the tree-structured file system, you must tell MS-DOS exactly where the file is located. MS-DOS needs to know the names of:

the drive
the directory
the file

If the file you want is located in the current directory, you don't have to supply a directory name. If the file is located in another directory (or subdirectory), you must give MS-DOS a roadmap (path) to follow.

Paths

A path is a list of directory names separated by backslashes. The path format is:

[d:] [\] [directory\]. . .[directory\][filename. extension]

The number of directories depends upon the complexity of your organization. Separate each reference to a directory or subdirectory with a backslash. Separate filenames by backslashes also.

The path begins in the root directory or the current directory. Start at the top of the tree and work down.

To specify a path, follow these rules:

1. A path cannot contain more than 63 characters.
2. Precede the colon and the path name with the device name.
3. If you assign a path to a string variable, enclose the string (including the device name and the colon) in quotation marks.

Using the preceding tree-structured directory example, access the file APPENDIX.ONE as follows:

1. To specify the path, start at the top of the tree. Enter the device name followed by a colon. Then begin the path. If you begin the path with a backslash, the search starts in the root directory. Otherwise, the search starts in the current directory.

   Example

   ```
   a:\Software\Project3\APPENDIX.ONE
   ```

2. To specify a path using a diskette in another drive or a fixed disk, assign the path to a string variable.

   Example

   ```
   path$ = "b:\SOFTWARE\PROJECT3\APPENDIX.ONE"
   ```

Paths can be used with the following BASIC commands:

BLOAD	KILL	OPEN
BSAVE	LOAD	RMDIR
CHAIN	MERGE	RUN
CHDIR	MKDIR	SAVE
FILES		

You can refer to the string variable when you want to locate the file.

Example

```
a:path$
```

Using the Screen

BASIC allows you to display text, special characters, points, lines, and complex shapes in color or monochrome. Text refers to alphabetic, characters, numbers, and special symbols in the character set.

Display Types

Four types of display monitors are available with the PC: one monochrome and three color monitors.

Monochrome display text appears on the screen as green on a black background. You can set parameters using a COLOR statement, to produce reverse-video, invisible, highlighted, or underscored characters on the monochrome screen.

The three types of color monitors include a 16-color display, a 256-color display, and a composite interface display. Table 2–2 illustrates the various screen modes, the number of colors available with each mode, and the types of displays available. Also included in Table 2–2 are the resolutions and number of screen pages for each mode.

Table 2–2. Mode Variations

Mode				Resolution	Page	Color	Screen Display			
							Mono-chrome	256-Color	16-Color	Com-posite
Monochrome Display			Text	80x25	1	Monochrome	o	x	x	x
Color Display	Elementary		Text	80x25	4	16 colors	x	o	o	x
						Monochrome				
				40x25	8	16 colors	x	o	o	o
						Monochrome				
			Graphic	320x200	1	4 colors	x	o	o	o
				640x200	1	Monochrome	x	o	o	x
	AD	Screen A	Text	80x25	8	16 colors	x	o	x	x
						Monochrome				
				40x25	8	16 colors	x	o	x	o
						Monochrome				
			Graphic	320x200	4	4 colors	x	o	x	o
				640x200	4	Monochrome	x	o	x	x
		Screen B	Graphic	320x200	2	256 colors	x	o	x	x
				640x200	2	16 colors	x	o	x	x
				320x400	1	256 colors	x	o	x	x
				640x400	1	16 colors	x	o	x	x

AD=Advanced o=permitted x=not permitted

Text and graphic modes can be used in medium or high resolution. High resolution text mode is an 80x25-character screen. Medium resolution text mode is a 40x25-character screen.

In graphic modes, resolution refers to the number of dots used within the screen. High resolution graphic mode is 640x200 screen dots. Medium resolution graphic mode is 320x200 screen dots.

Refer to the SCREEN statement for more information on color screen displays.

Text Mode

The text mode screen layout is shown in Figure 2-1.

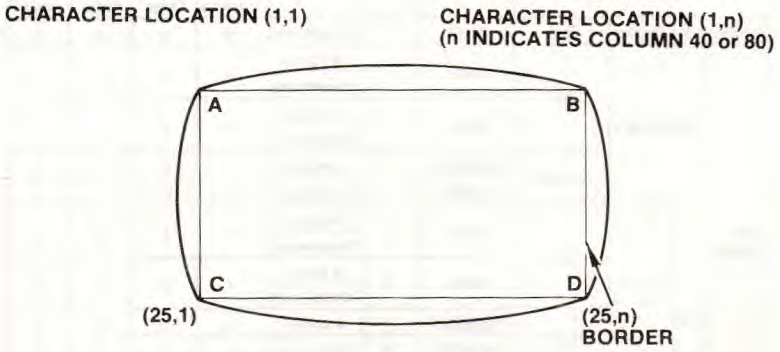


Figure 2-1. Text Mode Screen Layout

You may specify a screen of either 40 or 80 columns by using a WIDTH statement.

To specify the character location, use a LOCATE statement. The POS (0) and CSRLIN functions show the row and column position of a character on the screen. For example, the character C in Figure 2-1 is located in row 25, column 1. Note that the /G option gives you 25 lines (rows) instead of 24.

Usually, you display characters on the screen by using a PRINT statement. The cursor indicates where a character is to be displayed. Characters are output on a line from left to right. When you reach the end of a line, the character output continues on the following line. When you reach the last character position on the screen, the entire screen is scrolled up one line, leaving a blank line for your continued entry.

Each character on the screen has a foreground and a background. The foreground is the character itself. You may specify foreground and background colors for each character by using a COLOR statement. You may also specify blinking, invisible, highlighted, or underscored characters.

The color codes in the 16-color display are:

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

The frame around the screen in which no characters are displayed is the border. You may also specify the color of the border by using a COLOR statement.

In text mode, the following statements may be used to process text:

CLS	PRINT
COLOR	SCREEN
LOCATE	WIDTH
PALETTE	WRITE

The following functions are provided:

CSRLIN
POS
SCREEN
SPC
TAB

If a color display is connected, the screen is a 4-page or 8-page screen in the 80x25 and 40x25 modes, respectively.

In the advanced text mode, an 8-page screen is provided for both 80x25 and 40x25 modes.

You may specify a page to be displayed on the screen or a page you want to write to by using a SCREEN statement.

Graphic Modes

Graphic modes are only valid if a color display is connected. Graphic modes are elementary or advanced.

In each graphic mode, the following statements are available:

CIRCLE	PRESET
COLOR	PSET
DRAW	PUT
GET	SCREEN
LINE	VIEW
PAINT	WINDOW
PALETTE	

The POINT function is also available.

Elementary Mode

In this mode, you may select two types of resolution by using a SCREEN statement:

Medium Resolution–320x200 in four colors

1. In medium resolution, you can identify 320 horizontal dots and 200 vertical dots (Figure 2–2).

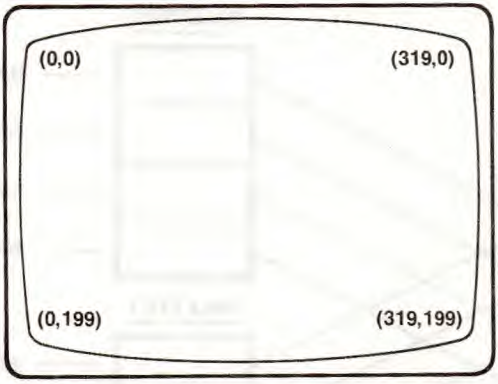


Figure 2-2. Elementary Mode Medium Resolution Screen

2. Color codes 0, 1, 2, and 3 are available for the medium resolution screen. The color codes are associated with colors via the COLOR statement palette (Figure 2-3). (This palette is not related to the PALETTE statement.)

3. Specify palette 0 or 1 using the COLOR statement.

4. You can also display text in the graphic modes. In medium resolution, character size is the same as 40x25 in the text mode and the color codes for foreground and background are 3 and 0, respectively.

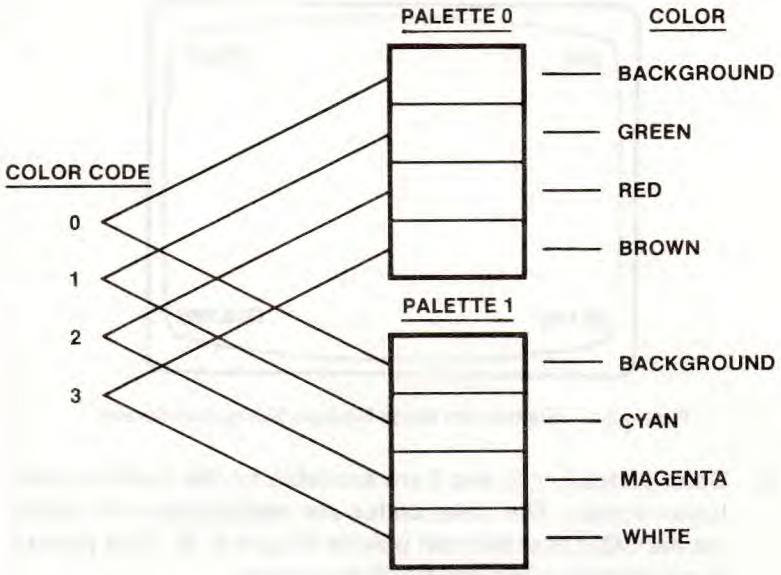


Figure 2-3. Medium Resolution Color Palette

High Resolution-640x200 in two colors

1. In high resolution, you can identify 640 horizontal dots and 200 vertical dots (Figure 2-4).
2. Only two colors are available in high resolution: black (0) and white (1).
3. When you display text characters in high resolution, character size is the same as for 80x25 in the text mode. The color codes for foreground and background are 1 and 0 (white on black).

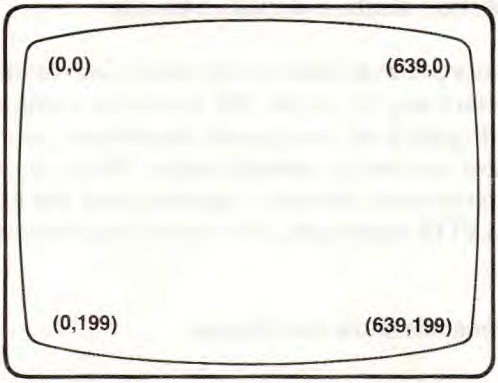


Figure 2-4. High Resolution Screen

Advanced Mode (Screen A). This advanced mode is the same as the elementary mode except for the additional number of pages available in advanced mode (refer to Table 2-2). Screen A can be used for text or graphics.

Advanced Mode (Screen B). You can select four types of resolution by using a SCREEN statement in this mode: two medium resolution modes and two high resolution modes. Screen B is used for graphics only.

Medium Resolution Mode-320x200, 320x400. As many as 256 colors can be handled in medium resolution mode. These colors are represented by the numbers 0 through 255 made up of eight red contrasts, eight green contrasts, and four blue contrasts. Any of these colors may be specified by using the LINE or CIRCLE statement (when a 256-color display is connected).

High Resolution Mode–640x200, 640x400

Only 16 colors are available in high resolution mode. However, you may select any 16 of the 256 colors by using a PALETTE statement (Figure 2–5). In a graphic statement, you must use a color register number to specify color. When you change the relationship between the color registers and the actual colors using a PALETTE statement, the colors displayed may change instantly.

Screen B does not allow text display.

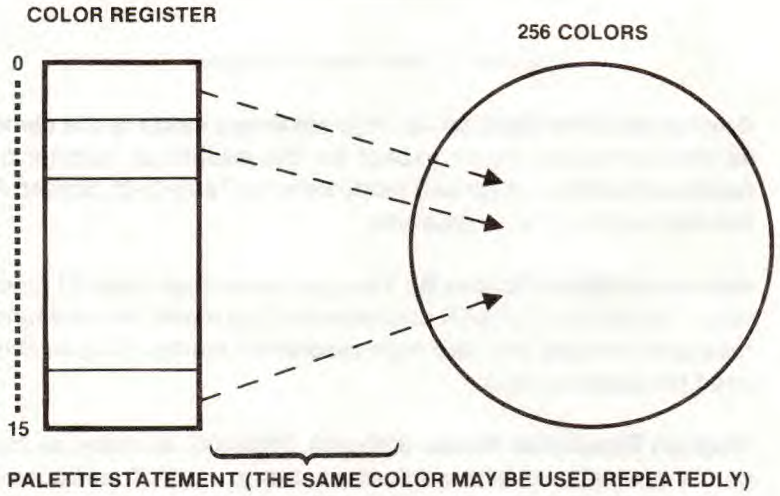


Figure 2–5. Palette Statement

PALETTE Statement

If you are using the 256-color display in a mode other than medium resolution for screen B, the color code you specify in a graphic statement or a COLOR statement is treated as a color register number; therefore, a PALETTE statement affects the display.

Specification of Coordinates

To display a pixel (point) with a graphic statement, you must specify the location of the pixel. Generally, the location is specified using coordinate values in the form (x,y), where x indicates the horizontal position and y indicates the vertical position. This is called absolute form, and indicates an actual position of the pixel on the screen.

Relative form indicates the position of the pixel relative to the last pixel referenced. This form is called offset. The format is:

STEP (xoffset, yoffset)	(xoffset represents the horizontal offset position and yoffset represents the vertical offset position.)

The last referenced position depends on the result of each graphics statement executed.

Superimposed Mode

The text on screen A in the advanced mode may be overlapped (superimposed) on screen B. In the superimposed mode, you can manage screen A and screen B separately.

Use a SCREEN statement to specify the superimposed mode.

Advanced Input and Output Features

Additional I/O features include clock and sound.

Clock

You can set or read the time and date by using these system functions:

DATE$	A 10-character string in the format mm-dd-yyyy
TIME$	An 8-character string in the format hh:mm:ss

Sound

You can produce sounds by using the following statements:

BEEP	Generates a short beep.
SOUND	Produces a single sound of a specified frequency for a specified duration of time.
PLAY	Defines a character string that plays music.

Keyboard

Figures 2-6 through 2-8 show the keyboards that are used with your PC. The keyboards contain the same keys, but key locations vary. In this manual, the keyboard in Figure 2-6 is used to illustrate the location and function of keys.

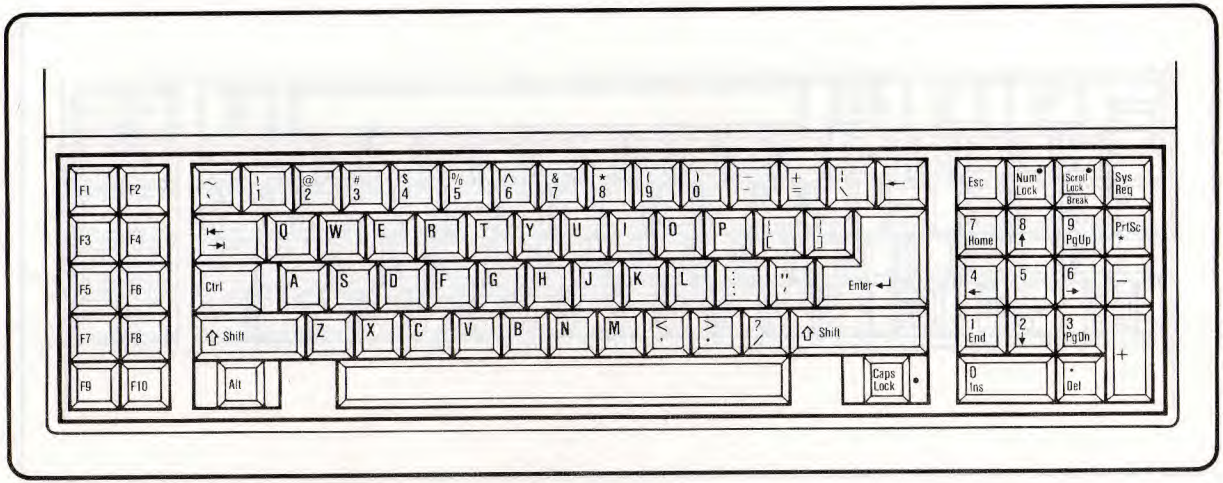


Figure 2-6. PC/IT Keyboard

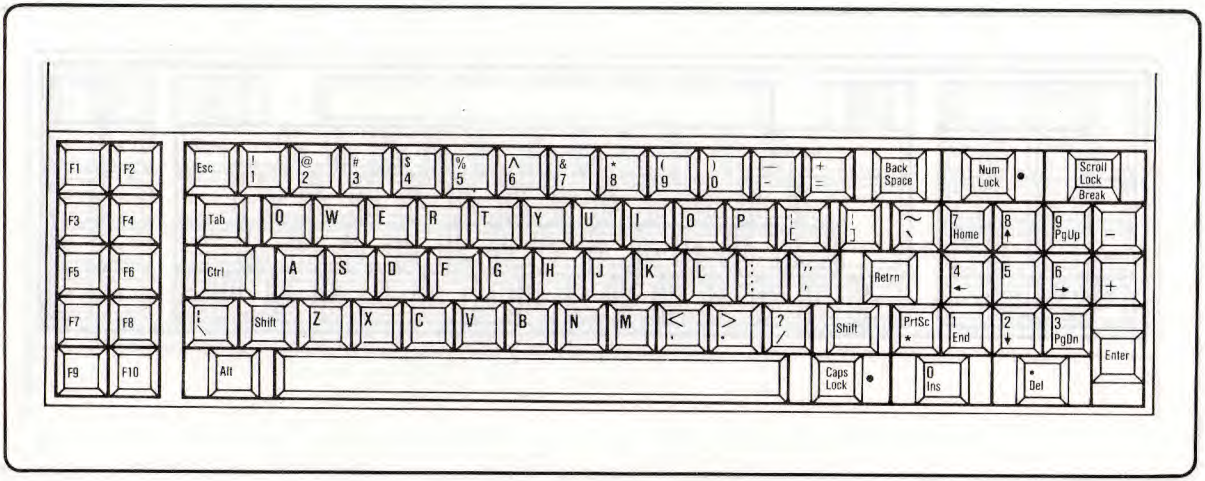


Figure 2-7. PC Keyboard

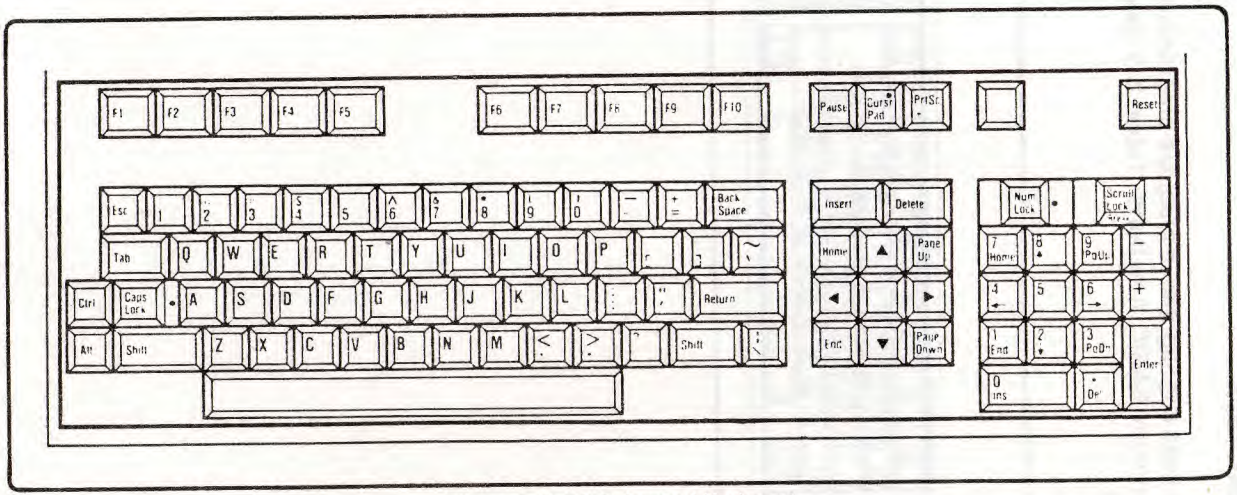


Figure 2–8. PC Enhanced Keyboard

Function Keys

Function keys (highlighted in blue) are programmed to activate some common BASIC functions. Table 2–3 describes these functions.

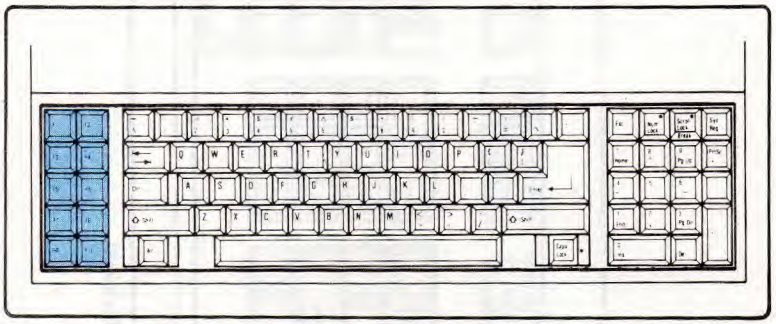
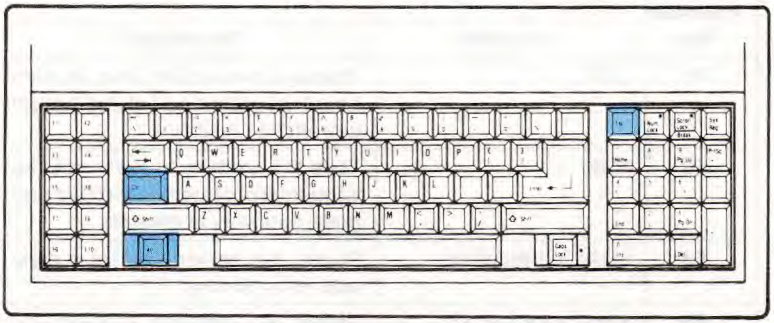


Table 2-3. Function Key Functions

Function Key	Function	Description
F1	LIST	Displays the existing lines, in numbered sequence, of the program currently in memory.
F2	RUN	Tells the system to carry out the sequence of statements of the program currently in memory.
F3	LOAD <filename>	Reads the named file from diskette into memory.
F4	SAVE <filename>	Reads the program currently in memory and writes it to the diskette, using the specified filename.
F5	CONT	Restarts a program after it has been interrupted by the Ctrl Break key sequence.
F6	Device Name	Inserts ,"LPT1:" at the current cursor position. This device name is frequently used in commands such as OPEN and LIST.
F7	TRON	Turns trace on. Displays the line number of a program as it is run.
F8	TROFF	Turns trace off.
F9	KEY	Changes the function of the other function keys.
F10	SCREEN	Changes the program from graphic mode to text mode.

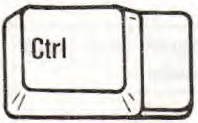
Special Function Keys



Special function keys are used to control some special functions of the computer. They are operational at all times, but may have different uses depending on the software package you are using.

Examples of the uses of these keys in BASIC follow.

The **Alt** key is also used for special functions. The **Alt** key combinations are described in this section.

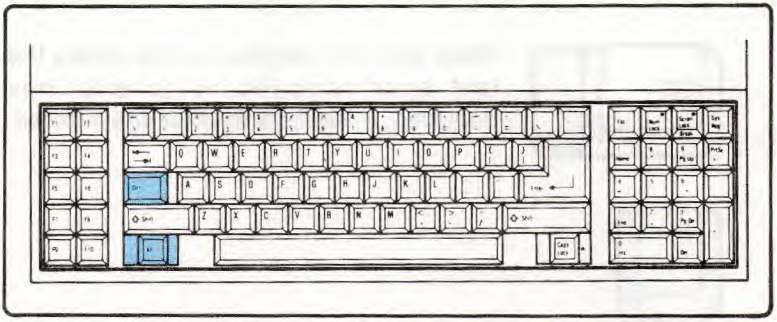


The **Ctrl** (control) key is always used with another key. The special functions of the **Ctrl** key are also described in this section.

Use the **Esc** (escape) key to erase the line you are currently editing or entering. If the erased line has been entered as a program line, it is not erased from memory by pressing **Esc**.

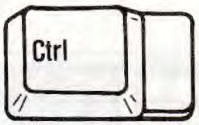
Control Keys

The keys (highlighted in blue) perform special editing functions when you press them simultaneously with other keys on the keyboard. Control key combinations are illustrated and described in this section.



Press the **Alt** key with a letter key to enter a BASIC keyword. Each letter key and corresponding keyword follows.

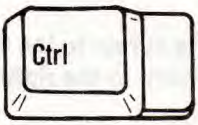
Letter	Keyword	Letter	Keyword
A	AUTO	M	MERGE
B	BSAVE	N	NEXT
C	COLOR	O	OPEN
D	DELETE	P	PRINT
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX$	U	USING
I	INPUT	V	VAL
K	KEY	W	WIDTH
L	LOCATE	X	XOR

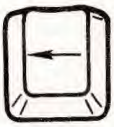


Press **Ctrl A** to display the line where the last error occurred. If no error has occurred, it displays the last line entered.

Press **Ctrl, Alt, Del** to reset the system (exit BASIC and return to MS-DOS).

Press **Ctrl B** to move the cursor back to the previous entry.



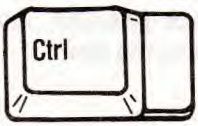


Press **Ctrl Back Space** to erase the character at the cursor position. Other data on the line shifts to the left.

Press **Ctrl Scroll Lock/Break** to exit AUTO mode and return to command level. The line being edited is not saved, but it is still displayed on the screen.

If you press **Ctrl Break** while a program is running, you can restart the program by pressing the **F5** key.

Press **Ctrl E** to erase the contents of the line from the cursor position to the end of the line.



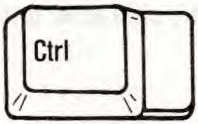
Press **Ctrl F** to move the cursor to the first character in the next word to the right.

Press **Ctrl G** to cause a beep.

Press **Ctrl H** to erase the character left of the cursor position.

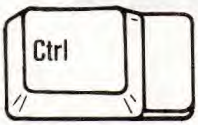
Press **Ctrl I** to move the cursor eight positions forward, leaving eight null spaces. (The **Tab** key has the same function.)

Press **Ctrl J** to send a line feed command to the system (move the cursor to the first column position of the next line).



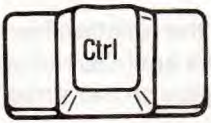
Press **Ctrl K** to move the cursor to home position (the top left corner of the screen).

Press **Ctrl L** to erase the screen and return the cursor to the home position in the upper left corner of the screen. **Ctrl Home** performs the same function as the **Ctrl L** and both perform the same function as the CLS statement.



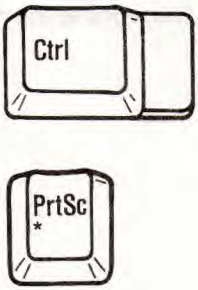
Press **Ctrl M** to move the cursor to the first column position of the next line.

Press **Ctrl N** to move the cursor to the end of the line.



Press **Ctrl Num Lock** to interrupt a function. To continue, press any alphanumeric or cursor control key.

Use **Ctrl Num Lock** to halt printing or program listing temporarily.



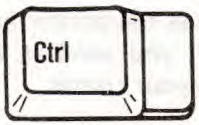
Press **Ctrl PrtSc** to use the printer as a system log. Any text you send to the screen is also sent to your printer.

To cancel this function, press **Ctrl PrtSc** again.

Press **Ctrl R** to insert a character at the cursor position and shift the data on the same line to the right.

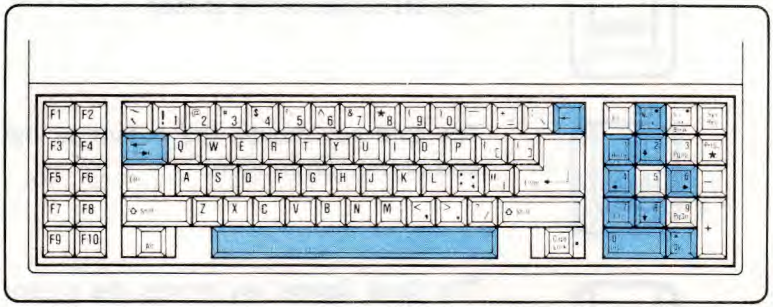
Press **Ctrl T** to erase the function key display. Press these keys again to redisplay the function key display on the screen.

If the screen width is set to 40 columns, press **Ctrl T** once to display function keys 6 through 10. Press **Ctrl T** again to erase the function key display.

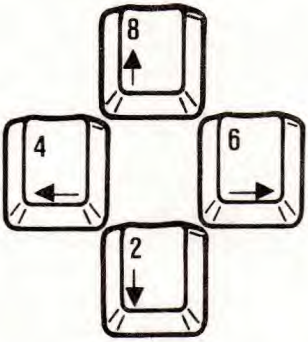


Press **Ctrl W** to delete the word at the cursor position.

Editing Keys



Most of the editing keys are on the numeric keypad. The **NUM LOCK** key controls the functions of the keypad. When you press **NUM LOCK** and the indicator lights, numbers are selected. When the indicator is off, editing functions are inplemented.

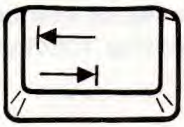


The four arrow keys are used to move the cursor around on the screen. Each press of an arrow key moves the cursor one character position to the left or right, or one line up or down.

The **Home** key moves the cursor to the top left corner of the screen.

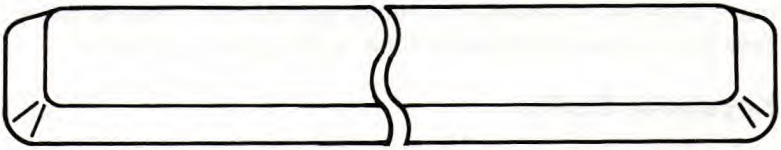
The **End** key moves the cursor to the end of the current line.

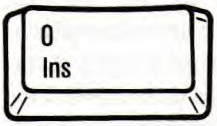
The **Back Space** key moves the cursor back one space, erasing any character the cursor passes over.



The **Tab** key moves the cursor forward to the next tab stop position (every eight positions).

The cursor can also be moved to the right with the space bar. The space bar erases any characters the cursor passes over.





The **Ins** (insert) key is used to enter insert mode. That is, data entered is inserted at the cursor position, shifting the previously entered data on that line to the right.

The **Del** (delete) key is used to delete characters or blocks of data.

Key Cycling

Each key has an automatic repeat function. If you hold a key down for more than a half second, the system repeats (cycles) that character at a rate of 10 times per second. If two or more keys are pressed at the same time, cycling does not occur.

Keystroke Buffer

Data entered at a rate faster than the system can act on it is stored in the keystroke buffer. The system acts on information sequentially, storing data as it is received and retrieving data when it is ready to be acted upon. Therefore, you do not need to slow down or stop typing if characters are not displayed on the screen immediately.

Chapter 3. BASIC Commands and Statements

Table

Chapter 3. BASIC Commands and Statements

Each BASIC command and statement is described in this chapter.

Descriptions are formatted as follows:

Format Shows the correct format for the instruction. Format notation appears as follows:

COMMAND [<filename>][options]

Purpose States the purpose of the instruction.

Remarks Describes in detail how the instruction is used.

Example Shows sample programs or program segments that demonstrate the use of the instruction.

AUTO

Format

AUTO [<line number>[,<increment>]]

Purpose

To generate a line number automatically after every carriage return.

Remarks

AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, entering a return immediately after the asterisk saves the line and generates the next line number.

AUTO is terminated by pressing **Ctrl Break**. The last line entered is not saved.

After **Ctrl Break** is pressed, BASIC returns to command level.

Examples

AUTO 100,50	Generates line numbers 100, 150, 200 . . .
AUTO	Generates line numbers 10, 20, 30, 40 . . .

BEEP

Format

```
BEEP
```

Purpose

The BEEP statement sounds the speaker at approximately 1000 Hertz for 1/4 second.

Remarks

Both BEEP and PRINT CHR$(7) produce the same effect.

Example

```
100 FOR I=1 TO 5
110 BEEP
120 FOR J=0 TO 100:NEXT J
130 NEXT I
140 END
```

BLOAD

Format

BLOAD <filespec>[,<offset>]

Purpose

The BLOAD statement allows a file to be loaded anywhere in user memory.

Remarks

<filespec> is a valid string expression containing the device and filename. If the device name is omitted, the device in current use is assumed. When <filespec> does not agree with the rules concerning filenames, a "Bad file name" error message is displayed and loading is discontinued.

<offset> is a valid numeric expression returning an unsigned integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement at which loading is to start.

If <offset> is omitted, the offset specified at BSAVE is assumed. (The file is loaded into the same location from which it was saved.)

If <offset> is specified, a DEF SEG statement should be executed before the BLOAD. When <offset> is given, BASIC assumes the user wants to BLOAD at an address other than the one saved.

With this statement, the memory image file (machine language program, screen data, etc. as specified in <filespec>) is loaded into user memory.

BLOAD and BSAVE are useful for loading and saving machine language programs. (Refer to the CALL statement.) However, use of BLOAD or BSAVE is not restricted to machine language programs. Any segment may be specified as the source or target for these statements using the DEF SEG statement.

CAUTION:

BLOAD does not perform an address range check. It is possible to use BLOAD anywhere in memory. Do not use BLOAD over BASIC's stack, program, or variable area.

Example

```
100 'LOAD V-RAM data from "SCR"
110 DEF SEG=&HB800
120 BLOAD "SCR",0
```

Reference

BSAVE, CALL, DEF SEG

BSAVE

Format

BSAVE <filespec>, <offset>, <length>

Purpose

The BSAVE statement allows portions of the user's memory to be saved on the specified device.

Remarks

<filespec> is a valid string expression containing the device and filename.

<offset> is a valid numeric expression returning an unsigned integer in the range 0 to 65535. The offset number points to a memory location within the storage segment declared by the last DEF SEG command.

The memory location is found by adding the offset to the segment value.

<length> is a valid numeric expression returning an unsigned integer in the range 1 to 65535. This is the length of the memory image to be saved.

BLOAD and BSAVE are most useful for loading and saving machine language programs. (Refer to the CALL statement.) However, BLOAD and BSAVE are not restricted to machine language programs. Any segment may be specified as the source or target for these statements by using the DEF SEG statement.

If a disk drive is not specified with a device name, the drive in use is assumed.

If <filename> is less than one character or more than eight characters, a "Bad file name" error message is displayed and the save is aborted.

If <offset> is omitted, a "Syntax error" message is displayed and the save is aborted. A DEF SEG statement should be executed before the BSAVE. The last DEF SEG address is always used for the save.

If <length> is omitted, a "Syntax error" message is displayed and the save is aborted.

Example

```
100 DEF SEG=&HB00
120 BSAVE "SCR",0,&H8000
```

Reference

BLOAD, CALL, DEF SEG

CALL

Format

CALL <variable name>[(<argument list>[,<argument list>...])]

Purpose

The CALL statement is recommended for interfacing machine language programs with BASIC.

Remarks

<variable name> is the address of the starting point in memory of the subroutine being called.

<argument list> contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement transfers program control to a machine language subroutine at the memory location whose address is expressed by <variable name>.

Refer to USR for another way of interfacing machine language programs with BASIC.

Example

```
100 DEF SEG = &H800
110 BLOAD "Machine.exe",0 'load machine routine
120 '.
130 '.
140 '.
300 SUBO = 0
310 CALL SUBO(A,B,C)
320 '.
330 '.
```

Reference

DEF SEG, DEF USR, Appendix C

CHAIN

Format

```
CHAIN [MERGE] "<filename>"[,[<line number exp>]
[,ALL] [,DELETE <range>] ]
```

Purpose

To call a program and pass variables to that program from the current program.

Remarks

<filename> is the name of the program being called.

<line number exp> is the line number in the called program where program execution will begin. If it is omitted, execution begins at the first line.

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed.

```
CHAIN "Prog100",1000,ALL
```

If the MERGE option is included, a subroutine may be brought into the BASIC program as an overlay. That is, the current program can be merged with the called program. The called program must be an ASCII file if it is to be merged.

```
CHAIN MERGE "Overlay",1000
```

After an overlay is entered, it is usually best to delete it so a new overlay may be entered. To do this, use the DELETE option.

```
CHAIN MERGE "Overlay2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

Reference

COMMON

NOTE:

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

If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. Any DEF or DEF FN statement containing shared variables must be restated in the chained program.

CHDIR

Format

CHDIR "<path>"

where <path> is:

[d:][<directory>\]...<directory>

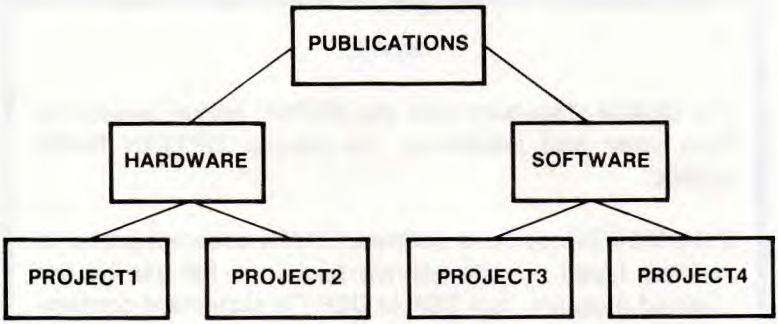
Purpose

To identify a directory that will become the current directory.

Remarks

<path> is formatted as a string expression and cannot exceed 63 characters.

The following tree-structured directory with the root directory named Publications is used as an example:



Examples

```
CHDIR "\"
```

This command is used to return the root directory, regardless of the current directory.

```
CHDIR "Hardware\Project2"
```

This command changes the current directory (Publications) to a file (Project2) which is located in a subdirectory (Hardware). Trace from the top of the tree to the bottom until the desired subdirectory is found.

```
CHDIR "\Hardware\Project1"
```

This command moves the current directory to a subdirectory (Project1).

```
CHDIR ".."
```

This command is the inverse of CHDIR. The string expression (. .) means, "Find the next-higher-level directory." For example, if you enter the command

```
CHDIR "Hardware\Project2"
```

followed by the command

```
CHDIR ".."
```

The current directory moves from the directory (Project2) to the next-higher-level directory (Hardware).

```
CHDIR "<device>:<directory>"
CHDIR "B:Trouble"
```

Current directories on one drive are changed on a drive other than the one in use.

CIRCLE

Format

CIRCLE (<xcenter>,<ycenter>),<radius> [,<color>
[,<start>,<end>[, <aspect>]]]

Purpose

The CIRCLE statement draws a circle with a center and radius indicated by the first of its arguments.

Remarks

The center coordinates can be specified in absolute or relative form.

<color> specifies the color of the line segment. The range depends on the graphic mode, as follows:

Mode			Range of <color>	Default Value
Elementary Mode		320x200	0 through 3	3
		640x200	0,1	1
Advanced Mode	Screen A	320x200	0 through 3	3
		640x200	0,1	1
	Screen B	320x [200,400]	0 through 255	Depends on <foreground> in COLOR statement
		640x [200,400]	0 through 15	

For the relationship between <color> and the actual colors, refer to the COLOR statement.

In the superimposed mode, the default value of <color> depends on <Sforeground> in the COLOR statement.

When a 256-color display is connected, colors may be changed using a PALETTE statement to change corresponding color codes (except in medium resolution for screen B).

The start and end angle parameters are radian arguments between –2 * pi and 2 * pi which allow you to specify where the drawing of the circle will begin and end. If the beginning or ending angle is negative (–0 is not permitted), the circle will be connected to the center point with a line and the angles will be treated as if they were positive. (Note that this is different from adding 2 * pi.)

Aspect ratio refers to the ratio of x radius to y radius. The default values are 5/6 for the 320x200 and 640x400 modes, 5/12 for the 640x200 mode, and 10/6 for the 320x400 mode.

If the aspect ratio is less than 1, the radius is given in x-pixels. If it is greater than 1, the radius is given in y-pixels. The standard relative notation may be used to specify the center point.

If the circle drawn is too large, parts of it may run off the screen. A process called line clipping will clip the circle at the boundaries of the screen viewing area. Points plotted outside the viewing area will not appear and will not wrap around.

Reference

COLOR, PALETTE

CLEAR

Format

```
CLEAR [,[<expression1>] [,<expression2>]]
```

Purpose

To set all numeric variables to zero, all string variables to null, close all open files, and (optionally), to set the end of memory and the amount of stack space.

Remarks

<expression1> is a memory location. If specified, this sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 512 bytes or one-eighth of the available memory, whichever is smaller.

Examples

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

CLOSE

Format

```
CLOSE [[#]<file number>[,[#]<file number>]...]
```

Purpose

To conclude I/O to a disk file.

Remarks

<file number> is the number under which the file was opened. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE command. The file may then be reopened using the same or a different file number. Likewise, that file number may now be reused to open any file.

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

The END, NEW, RESET, SYSTEM, or RUN statement without the R option always closes all disk files automatically. (STOP does not close disk files.)

Reference

Appendix A

CLS

Format

CLS [<screen identifier>]

Purpose

To erase the currently active screen page. (Refer to the SCREEN statement.)

Remarks

The CLS statement returns the cursor to home position in the upper left corner of the screen.

<screen identifier> is a number between 1 and 3. If no number is specified, the default is 1.

1 Erases all active pages.
2 Erases the current active page on screen A.
3 Erases the current active page on screen B.

CLS does not erase background or border colors set in a COLOR statement. Use the SCREEN statement to clear colors, graphics, and text.

The screen may also be cleared by pressing **Ctrl Home**. However, as with CLS, these keys do not erase border and background colors.

Ctrl L performs the same function as **Ctrl Home**.

Reference

SCREEN

COLOR (Text)

Format

COLOR [<foreground>] [,[<background>] [,<border>]]

Purpose

To set <foreground>, <background>, and <border> colors on the text screen.

Remarks

<foreground> is a numeric expression from 0 to 31 that specifies the color of the characters.

<background> is a numeric expression from 0 to 7 that specifies the color of the background.

<border> is numeric expression from 0 to 15 that specifies the color of the screen border.

The following colors are available for <foreground>:

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

When 16 is added to a color code, the character blinks. (For example, 17 specifies a blinking, blue character.)

Of color codes 0 through 7, only one color is used for <background>.

When a monochrome display is connected, the following codes are available for <foreground>:

0	Black
1	Underscored white
2 through 7	White

As with a color display, adding 8 to a code highlights that character. For example, 10 is high-intensity white. (There is no high-intensity black.)

If 16 is added to a code, the character blinks. A blinking black character is 16. A blinking, high-intensity white character is 31.

The following codes are available for <background> on a monochrome display:

0 through 6	Black
7	White

White (7) only appears as a background color when the foreground color is black (0), gray (8), blinking black (16), or blinking gray (24). Black characters on a white background produce a reverse-video display.

Black (0), gray (8), blinking black (16), and blinking gray (24) foregrounds are visible only against a white (7) background. These colors are invisible against a black (0) background.

Other combinations of foreground and background colors produce white on black.

Example

```
25 COLOR 9, 1, 10
```

The colors in this example produce a light blue foreground against a blue background with a light green border.

NOTE:

If the value of a parameter exceeds 255, an "Illegal function call" error occurs. In this case, the old value remains.

Any parameter may be omitted. If a parameter is omitted, the old value remains.

COLOR (Graphics)

Format

COLOR [<background>] [,[<palette>]]
COLOR [<background>] [,[<foreground>]]

Purpose

To set colors to be used in graphic mode.

Remarks

<background> is a numeric expression that specifies the color of the background. Values 0 through 15 are allowed. For the relationship between these values and the actual colors, refer to COLOR (Text). On screen B, only black (0) is permitted for <background>.

<palette> is a numeric expression used to select a color palette.

<foreground> is a numeric expression that specifies the color of the foreground.

The colors to be selected by each palette for elementary and advanced screen A, medium resolution, are:

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

If <palette> is an even number, palette 0 is selected; if it is an odd number, palette 1 is selected.

The COLOR statement has no effect on high resolution in elementary and advanced screen A.

<foreground> is valid only for advanced screen B. In the high-resolution mode, <foreground> is a numeric expression from 0 to 15. For the relationship between color codes and the actual colors, refer to COLOR (Text).

In the screen B medium-resolution mode, <foreground> is a numeric expression from 0 to 255. For the relationship between the color codes and the actual colors, refer to PALETTE.

Any parameter may be omitted. If a parameter is omitted, the old value remains.

If the value of a parameter exceeds 255, an "Illegal function call" error occurs. In such a case, the old value remains.

Example

```
5 SCREEN 1
10 COLOR 9,1
```

Reference

COLOR (Text), PALETTE

COLOR (Superimposed)

Format

COLOR [<foreground>] [,[<background>] [,[<border>],<Sforeground>]]

Purpose

To assign colors to text screen A or graphic screen B in the superimposed mode.

Remarks

<foreground>, <background>, and <border> are used to set colors of text screen A. See COLOR (Text).

<Sforeground> specifies the foreground color of screen B.

<border> is always black.

In the superimposed mode, if a dot overlap occurs, the dot on screen A is given priority. Note that screen B is visible only when <background> for text screen A is black (0) or gray (8).

NOTE:

The background color of screen B is always black.

Reference

COLOR (Text)

COM

Format

```
COM (<port number>) ON
COM (<port number>) OFF
COM (<port number>) STOP
```

Purpose

To prepare for trapping from a communication port.

Remarks

<port number> is the communication port number (1 or 2).

Incoming communications can either be trapped (ON), ignored (OFF), or stopped (STOP).

After COM ON is set, the ON COM statement traps all incoming communications in the port specified in the ON COM statement.

When COM OFF is used, all incoming communications are not trapped and are ignored.

The COM STOP statement does not allow trapping, but holds the incoming communications until the COM ON statement is used, to allow trapping of the communications that have been stopped.

COMMON

Format

```
COMMON <list of variables>
```

Purpose

To pass variables to a chained program.

Remarks

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, but it is recommended that they appear at the beginning.

Do not use the same variable in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example

```
100 COMMON A,B,C,D( ),G$
110 CHAIN "Prog3", 10
```

Reference

CHAIN

CONT

Format

```
CONT
```

Purpose

To continue program execution after **Ctrl Break** has been pressed, or a STOP or END statement has been executed.

Remarks

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

CONT is usually used with the STOP command for debugging. When program execution is stopped, intermediate values may be examined and changed by using direct mode statements. Resume execution by using CONT or a direct mode GOTO (which resumes execution at a specified line number). CONT may also be used to continue execution after an error.

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

Reference

END, STOP

DATA

Format

DATA <list of constants>

Purpose

To store numeric and string constants accessed by READ statements.

Remarks

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants (separated by commas) as will fit on a line. Any number of DATA statements may be used in a program.

READ statements access DATA statements in order (by line number).

The data may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point, or integer. (Numeric expressions are not allowed in the list.) String constants in DATA statements must be surrounded by quotation marks if they contain commas, colons, or significant leading or trailing spaces.

The variable type (numeric or string) in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by using the RESTORE statement.

Reference

READ, RESTORE

DEF

Format

DEF <type> <range of letters>...

where <type> is:

INT, SNG, DBL, or STR

Purpose

To declare variable types as integer, single precision, double precision, or string.

Remarks

A DEF statement declares that the variable names beginning with the letter specified will be that variable type. A type declaration character always takes precedence over a DEF statement.

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

Examples

10 DEFDBL L-P	All variables beginning with the letters L, M, N, O, and P are double precision variables.
10 DEFSTR A	All variables beginning with the letter A are string variables.
10 DEFINT I-N, W-Z	All variables beginning with the letters I, J, K, L, M, N, W, X, Y, Z are integer variables.

DEF FN

Format

DEF FN<name>[(<parameter list>)] = <function definition>

Purpose

To define and name a function written by the user.

Remarks

<name> must be a legal variable name. Preceded by FN, this name becomes the name of the function.

<parameter list> is comprised of the variable names in the function definition that are to be replaced. Items in the list are separated by commas.

<function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names included in this expression only define the function; they do not affect program variables of the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values in the function call.

In BASIC, user-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines can be called. If a function is called before it has been defined, an "Undefined user function" error occurs.

DEF FN is illegal in the direct mode.

Example

```
.
.
.
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
.
.
.
```

(Line 410 defines the function FNAB. The function is called in line 420.)

DEF SEG

Format

```
DEF SEG [=<address>]
```

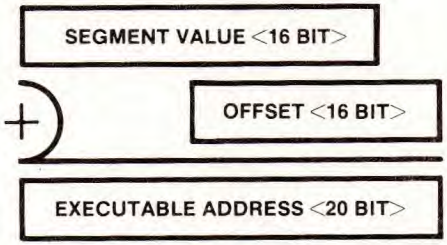
Purpose

To define the current storage segment.

Remarks

<address> is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

The address specified is saved for use as the segment required by BLOAD, BSAVE, PEEK, POKE, VARPTR, USR, and CALL statements.



If the address is omitted, the storage segment is set to BASIC's data segment. This is the initial default value.

If the address is included, it should be based on a 16-byte boundary.

For example, if <address> is set at &H800, the starting storage address becomes &H8000.

BASIC does not check to ensure that the result of the segment value plus the offset value is a valid address.

DEF and SEG must be separated by a space. Otherwise, BASIC interprets the statement DEFSEG = 100 to mean, "Assign the value 100 to the variable DEFSEG."

Any value entered outside this range results in an "Illegal function call" error. The previous value is retained.

Example

```
100 PRINT
110 DEF SEG=&HB800
120 DT=&H55
130 FOR AD=0 TO 32000 STEP 2
140 POKE AD, DT
150 NEXT AD
160 END
```

DEF USR

Format

```
DEF USR [<digit>]=<integer expression>
```

Purpose

To specify the starting address of an assembly language subroutine.

Remarks

<digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is specified. If <digit> is omitted, DEF USR0 is assumed.

<integer expression> is the starting address of the USR routine.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, allowing access to as many subroutines as necessary.

Example

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
.
.
.
```

Reference

Appendix C

DELETE

Format

```
DELETE [<line number>[-[<line number>]]]
DELETE [<line number>-]
```

Purpose

To delete program lines.

Remarks

BASIC always returns to command level after a DELETE command is executed. If <line number> does not exist in the program, an "Illegal function call" error occurs.

Examples

DELETE 40	Deletes line 40.
DELETE 40-100	Deletes lines 40 through 100 (inclusive).
DELETE -40	Deletes all lines up to and including line 40.
DELETE 40-	Deletes lines 40 through the end of the program.

CAUTION:

The DELETE command will erase the entire program if you do not specify a line number or range of line numbers.

DIM

Format

```
DIM <list of subscripted variables>
```

Purpose

To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks

If an array variable name is used without a DIM statement, the maximum value of its subscripts is assumed to be 10. If a subscript greater than the maximum specified is used, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0 (unless otherwise specified with the OPTION BASE statement).

The DIM statement sets all elements of the specified arrays to an initial value of zero.

Example

```
 5 DATA "A", "B", "C", "D", "E", "F", "G", "H", "I", "J"
10 DIM B$(10)
20 FOR I=1 TO 10
30 READ B$(I)
40 NEXT I
.
.
.
```

Reference

OPTION BASE

DRAW

Format

DRAW "<command string>"

Purpose

To draw figures on the screen.

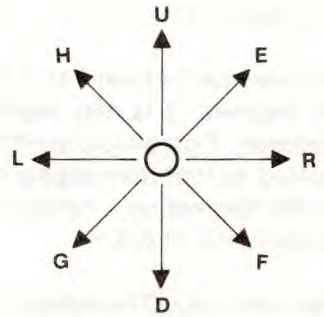
Remarks

A shape is defined using the command string. This shape is displayed when the DRAW statement is executed. BASIC interprets the command string letter-by-letter.

A character within a command string, optionally followed by one or more arguments, is a movement command.

Each of the following movement commands begin movement from the current graphic position. This is usually the coordinate of the last graphic point plotted with another movement command (LINE or PSET).

U (<n>)	Move up (scale factor *n) points.
D (<n>)	Move down.
L (<n>)	Move left.
R (<n>)	Move right.
E (<n>)	Move diagonally up and right.
F (<n>)	Move diagonally down and right.
G (<n>)	Move diagonally down and left.
H (<n>)	Move diagonally up and left.



DIRECTION OF MOVEMENT

The <n> in these commands indicates the movement distance. The actual points moved is <n>, adjusted to the scale factor set by the S command.

M<x,y> Move absolute or relative. If x is preceded by + or –, x and y are added to the current graphics position and connected to the current position with a line. Otherwise, a line is drawn to point x,y from the current position.

The spacing of points in horizontal, vertical, and diagonal movements is influenced by the screen aspect ratio (screen width divided by screen height).

The following characters may precede any of the movement commands:

B Move, but don't plot any points.

N Move, but return to original position.

A<n> Set angle <n>.

<n>may range from 0 to 3 (0 is 0 degrees, 1 is 90 degrees, 2 is 180 degrees, and 3 is 270 degrees). Figures rotated 90 or 270 degrees are scaled so that they appear the same size as 0 or 180 degrees on a monitor with the standard aspect ratio of 4/3.

C<n> Set color <n>. (The default is described in the CIRCLE statement.)

S<n> Set scale factor <n>.

<n> may range from 1 to 255 and 1/4 of this value becomes the scale factor. For example, the scale factor for 1 is 1/4. The scale factor is multiplied by the distance specified with U, D, L, R, E, F, G, H, or relative M commands to get the actual distance traveled.

X<string> Execute a substring (not supported by the BASIC compiler).

This command allows you to execute a second substring from a string, much like GOSUB in BASIC. One string may execute another, which executes a third, and so on.

TA<n>: Turn angle <n>.

<n> can range from −360 to +360. The angle turns counter-clockwise if <n> is positive and clockwise if <n> is negative. If values outside the range are entered, an "Illegal function call" occurs.

P<paint>, <boundary>:

Set figure color to <paint> and border color to <boundary>.

<paint> can range from 0 to 3. In medium resolution, the color is defined by the color statement from the current palette. In high resolution, 0 is black and 1 is white.

<boundary> can be in the range from 0 to 3. If you do not specify <paint> and <boundary>, an error occurs.

Numeric arguments can be constants, such as 123 or "=variable;", where variable is the name of a variable.

The semicolon, besides being used with variables, must be written in the X command. In the standard commands defined in the argument, the semicolon can also be used to separate commands.

Examples

```
100 CLS
110 SCREEN 1
120 H=199:W=319
130 DRAW "BM0,0"
140 WHILE H>=1
150 DRAW "R=W;D=H;L=W;U=H;"
160 HO=H/8:WO=W/8
170 DRAW "BR=WO;BD=HO;"
180 H=H*3/4
190 W=W*3/4
200 WEND
210 END
```

```
 5 SCREEN 1
10 DRAW "U50R50D50L50"        Draw a box.
20 DRAW "BE10"                Move up and right into the box.
30 DRAW "P1,3"                Paint interior.
```

```
 5 SCREEN 1
10 FOR D=0 TO 360             Draw some spokes.
20 DRAW "TA=D;NU50"
30 NEXT D
```

Reference

CIRCLE, COLOR

EDIT

Format

```
EDIT <line number>
EDIT .
```

Purpose

To enter edit mode at the specified line.

Remarks

In edit mode, it is possible to change portions of a line without retyping the entire line. BASIC enters the line number of the line to be edited, then leaves a space for an edit mode subcommand.

<line number> is the program line number to be edited. If there is no such line, the "Undefined line number" error message is displayed.

If you have just entered a line and you want to change it, use

```
EDIT .
```

to enter edit mode at the current line. This command can be used when an error message appears, or immediately following a STOP command.

Use the LIST command to display the contents of a program.

Reference

STOP

END

Format

```
END
```

Purpose

To terminate program execution, close all files, and return to command level.

Remarks

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break" message. Use of an END statement at the end of a program is optional.

After an END statement is executed, BASIC returns to command level.

Example

```
520 IF K>100 THEN END ELSE GOTO 20
```

ENVIRON

Format

ENVIRON <parm>

Purpose

To modify parameters in BASIC's environment string table. For example, the ENVIRON statement may be used to change the PATH parameter for a child process or to pass parameters to a child by inventing a new environment parameter.

Remarks

<parm> is a valid string expression containing the new environment string parameter.

<parm> must be of the form <parm id>=<text> where <parm id> is the name of the parameter and <text> is the new parameter text.

<parm id> must be separated from <text> by " = " or " ", such as "PATH = ". ENVIRON takes everything to the left of the first equal sign or blank as the <parm id>. The first character after <parm id> that is not an equal sign or a blank is taken as <text>.

If <text> is a null string or consists only of ";" (a single semicolon, such as "PATH=;"), then the parameter (including <parm id>=) is removed from the environment string table and the table is compressed.

If <parm id> does not exist, then <parm> is added at the end of the environment string table.

If <parm id> exists, it is deleted, the environment string table is compressed, and the new <parm> is added at the end.

If <parm> is not a string, a "Type mismatch" error occurs.

Examples

Unless changed by the MS-DOS PATH command, the environment string table is empty. The following BASIC statement creates a default path to the \DOS directory on disk C:

```
ENVIRON "PATH=C:\DOS"
```

You can now access MS-DOS external commands in the \DOS directory, even if you used a SHELL statement to change to another directory.

A new parameter may be added:

```
ENVIRON "MODE FILE=C:\PGMS"
```

The environment string table now contains:

```
PATH=C:\DOS
FILE=C:\PGMS
```

The PATH may be changed to a new value, for example:

```
ENVIRON "PATH=C:\SALES;C:\ACCOUNTING"
```

The environment string table now contains:

```
FILE=C:\PGMS
PATH=C:\SALES;C:\ACCOUNTING
```

The FILE parameter can be removed from the environment string table, as follows:

```
ENVIRON "FILE="
```

The environment string table now contains:

```
PATH=C:\SALES;C:\ACCOUNTING
```

If the environment string table is full and no more space can be allocated, an "Out of memory" error occurs.

Reference

ENVIRON$, SHELL

ERASE

Format

```
ERASE <list of array variables>
```

Purpose

To eliminate arrays from a program.

Remarks

Arrays may be redimensioned after they are erased or previously allocated array space may be used for other purposes. If an attempt is made to redimension an array before erasing it, a "Duplicate definition" error occurs.

Example

```
.
.
.
450 ERASE A, B
460 DIM B(99)
.
.
.
```

ERROR

Format

ERROR <integer expression>

Purpose

To simulate the occurrence of a BASIC error and allow error codes to be defined by the user.

Remarks

<integer expression> must be greater than 0 and less than 255. If <integer expression> equals an error code already in use by BASIC (Appendix F), the ERROR statement simulates the occurrence of that error and the corresponding error message is displayed.

To define your own error code, <integer expression> must be greater than any of BASIC's error codes. (Use the highest available values, so compatibility is maintained when more error codes are added to BASIC.) User-defined error codes may be conveniently handled in an error trap rountine.

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message "Unprintable error." Execution of an ERROR statement for which there is no error trap routine causes an error message and the execution is halted.

Example 1

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

or

```
Ok
ERROR 15          (You type this line.)
String too long   (BASIC displays this line.)
Ok
```

Example 2

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "What is your bet"; B
130 IF B>5000 THEN ERROR 210
.
.
.
400 IF ERR=210 THEN PRINT "House limit is $5000"
410 IF ERL=130 THEN RESUME 120
.
.
.
```

Reference

ON ERROR GOTO

FIELD

Format

FIELD [#] <file number>, <field width> AS <string variable>. . .

Purpose

To allocate space for variables in a random file buffer.

Remarks

To get data out of a random buffer after a GET command or enter data before a PUT command, a FIELD statement must have been executed.

<file number> is the number under which the file was opened. <field width> is the number of characters to be allocated to <string variable>.

If the total number of bytes allocated in a FIELD statement exceeds the record length specified when the file was opened, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

This example allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not put any data in the random file buffer. (Refer to LSET/RSET and GET.)

Reference

GET, LSET/RSET, PUT, Appendix A

NOTE:

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

FILES

Format

```
FILES ["<filespec>"]
```

Purpose

To display the names of files on the disk.

Remarks

<filespec> is a string expression used to specify the file.

If <filespec> contains only the filename, all files using that name are displayed. If <filespec> is omitted, all the files on the current drive are listed. The filename is a string expression which may contain question marks to match any character in the filename or extension. An asterisk as the first character of the filename or extension will match any file or extension.

If the disk drive is specified in <filespec>, the files in that drive are displayed. If the disk drive is not specified, the drive in use is assumed.

FILES now allows path names. The directory for the specified path is displayed. If an explicit path is not given, the current directory is assumed. To list all filenames on drive B, enter FILES "B:" (In previous versions, FILES "B:*.*" listed the filenames on drive B.)

To list subdirectories, enter <DIR> following the directory name.

Examples

`FILES "\Sales"`	Displays the directory entry Sales.
`FILES "\Sales\Mary\"`	Displays all files in Mary's directory.

FOR. . .NEXT

Format

```
FOR <variable>=x TO y [STEP z]
.
.
.
NEXT [<variable>] [<variable>...]
```

where x, y, and z are numeric expressions.

Purpose

To allow a series of instructions to be performed in a loop a given number of times.

Remarks

<variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter.

Program lines that follow the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP and checked to see if its value is now greater than the final value (y). If not, BASIC returns to the statement after the FOR statement and the process is repeated. If the value is greater, program execution continues with the statement following the NEXT statement. This is a FOR. . .NEXT loop.

If STEP is not specified, the increment is assumed to be 1. If STEP is negative, the final value of the counter is less than the initial value. The value of the counter decreases by increments each time through the loop until the counter value is less than the final value.

Nested Loops. FOR. . .NEXT loops may be nested; that is, a FOR. . .NEXT loop may be placed within another FOR. . .NEXT loop. When loops are nested, each loop must have a unique variable name. The NEXT statement for the inside loop must appear before the NEXT statement for the outside loop. If nested loops have the same end point, one NEXT statement may be used for all of them.

Variables in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is displayed and program execution is terminated.

Examples

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
    1 20
    3 30
    5 40
    7 50
    9 60
Ok
```

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

In this example, the loop is not executed because the initial value of the loop exceeds the final value.

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

In this example, the loop is executed 10 times. The final value for the loop variable is always set before the initial value is set.

GET (Files)

Format

GET [#]<file number> [, <record number>]

Purpose

To read a record from a random disk file into a random buffer.

Remarks

<file number> is the number under which the file was opened. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The GET statement allows record numbers in the range 1 to 16,777,215. Therefore, large files with short record lengths can be accommodated.

Reference

Appendix A

NOTE:

After a GET statement, INPUT# and LINE INPUT# will read characters from the random file buffer.

GET (For COM Files)

Format

```
GET [#]<file number>,<n bytes>
```

Purpose

To get the permissible fixed-length I/O for communication files.

Remarks

<file number> is an integer expression returning a valid file number.

<n bytes> is an integer expression returning the number of bytes to be transferred into or out of the file buffer. The byte number cannot exceed the value set by the LEN option in the OPEN COM statement.

Reference

ON COM, OPEN COM

GET (Graphics)

Format

```
GET <array name>
GET (<x1>,<y1>)-(<x2>,<y2>),<array name>
```

Purpose

To read points from an area of the screen.

Remarks

(<x1>, <y1> and (<x2>, <y2>) are the coordinates of the opposite corners of the screen area.

GET reads the colors of the points in the specified screen area into the array. A rectangular area of the screen has opposite corner points (<x1>, <y1>) and (<x2>, <y2>). Using the B option in the LINE statement gives the same results.

The purpose of the array is to hold the image in a specified area without concern for precision, but it must be in numeric form. Size is calculated as follows:

```
<array size> = (4 + INT ((x*M + 7)/8)*Z*y + A-1)/A
```

Where x is the number of horizontal pixels, y is the number of vertical pixels, A is the number of bytes in the array element, and M*Z is the number of bits necessary to represent one pixel depending on the graphic mode.

The values of M and Z are shown in the following table.

Table 3-1

Graphic Mode			M	Z
Elementary		320x200	2	1
		640x200	1	1
Advanced	Screen A	320x200	2	1
		640x200	1	1
	Screen B	320x200	2	4
		640x200	1	4
		320x400	2	4
		640x400	1	4

For example, the number of bytes required to make a 15 by 20 image in elementary medium resolution would be calculated as:

```
4+INT ((15*2+7)/8)*20
```

In other words, 84.

The number of bytes per array element depends on the precision used. The following rules apply:

Two bytes for an integer
Four bytes for single precision
Eight bytes for double precision

Thus, an integer array would have 42 elements.

Storage of screen information using GET is:

Two bytes giving the x dimension in bits
Two bytes giving the y dimension in bits
The data itself

If an integer array is used, it is possible to examine the x and y dimensions. The x dimension is in element 0 and the y dimension is in element 1 of the array.

However, integers are stored low byte first, but the data is transferred high byte first.

The data for each row of points in the rectangle is left- justified on a byte boundary. If the number of bits stored is less than a multiple of eight, the rest of the byte is filled with zeros.

Example

```
100 SCREEN 1,0: COLOR, 0,0
110 FOR I=1 TO 10
120 LINE (0,0)-(RND*319,RND*199), RND*2+1
130 NEXT
140 FOR I=1 TO 10
150 LINE(319,199)-(RND*319,RND*199), RND*2+1
160 NEXT
170 DIM A%(128)
180 GET (100,100)-(110,120), A%
190 END
```

Reference

PUT

GOSUB. . .RETURN

Format

```
GOSUB <line number>
.
.
.
RETURN
```

Purpose

To branch to and return from a subroutine.

Remarks

<line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program or from within another subroutine. Such nesting of subroutines is limited only by available memory.

RETURN statements in a subroutine cause BASIC to return to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement if a return at different points in the subroutine is necessary.

Subroutines may appear anywhere in the program, but the subroutine should be distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
10 GOSUB 40
20 PRINT "Back from subroutine"
30 END
40 PRINT "Subroutine ";
50 PRINT "in ";
60 PRINT "progress"
70 RETURN
RUN
Subroutine in progress
Back from subroutine
Ok
```

GOTO

Format

```
GOTO <line number>
```

Purpose

To branch (unconditionally) out of the normal program sequence to a specified line number.

Remarks

If <line number> is an executable statement, that statement and those following it are executed. If it is a nonexecutable statement, program execution proceeds to the first executable statement encountered after <line number>.

Example

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

IF

Format

```
IF <expression> THEN <line number or statements>
[ELSE <line number or statements>]

IF <expression> GOTO <line number>
[ELSE <line number or statements>]
```

Purpose

To make a decision regarding program flow based on the result returned by an expression.

Remarks

If the result of <expression> is not zero, THEN or GOTO is executed.

THEN may be followed by a line number for branching or one or more statements to be executed.

GOTO is always followed by a line number.

If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Program execution continues with the next executable statement.

Nesting of IF Statements. IF. . .THEN. . .ELSE statements may be nested. Nesting is limited only by the length of the line.

Examples

```
IF x>y THEN PRINT "Greater than" ELSE IF x<y
THEN PRINT "Less than" ELSE PRINT "Equal"
```

If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN.

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

will not print "A< >C" when A< >B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results (unless a statement with the specified line number had previously been entered in the indirect mode).

When using IF to test equality for a value that is the result of a floating point computation, the internal representation of the value may not be exact. Therefore, the test should be against a range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

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

This statement gets record number I (if I is not zero.)

```
100 IF(I<20) AND (I>10) THEN DB=1979-1:GOTO 300
110 PRINT "Out of range"
.
.
.
```

This statement initiates a test to determine whether I is greater than 10 and less than 20. If I is in this range, DB is calculated and program execution branches to line 300. If I is not in this range, execution continues with line 110.

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

This statement causes printed output to go to the terminal or line printer, depending on the value of the variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the terminal.

INPUT

Format

```
INPUT [;] ["<prompt string>";] <list of variables>
```

Purpose

To allow input from the terminal during program execution.

Remarks

When an INPUT statement is encountered, program execution pauses and a question mark is displayed. This indicates the program is waiting for data. If <prompt string> is included, the string is displayed before the question mark. The required data is then entered at the terminal.

To suppress the question mark, use a comma instead of a semicolon after the prompt string. For example, the statement.

```
INPUT "Enter birthdate", B$
```

Will display the prompt with no question mark.

If INPUT is immediately followed by a semicolon, the carriage return used to input data does not echo a carriage return/line feed sequence.

The data entered is assigned to the variables given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

Variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item input must agree with the type specified by the variable name. (Strings in an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the error message "?Redo from start." Input values are not assigned until an acceptable response is entered.

Examples

```
10 INPUT x
20 PRINT x "Squared is" x^2
30 END
RUN
? 5
5 Squared is 25
Ok
```

(The 5 was entered by the user in response to the question mark.)

```
LIST
10 PI=3.14
20 INPUT "What is the radius";R
30 A=PI*R^2
40 PRINT "The area of the circle is";A
50 PRINT
60 GOTO 20
Ok
RUN
What is the radius? 7.4
The area of the circle is 171.9464

What is the radius?
.
.
.
```

INPUT#

Format

```
INPUT# <file number>,<variable list>
```

Purpose

To read data from a sequential disk file and assign it to program variables.

Remarks

<file number> is the number used when the file was opened.

<variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.)

Items in the file should appear just as they would if data were being typed in response to an INPUT statement. When using numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered is assumed to be the start of a number. The number terminates with a space, carriage return, line feed, or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are ignored. The first character encountered is assumed to be the beginning of a string item. If this character is a quotation mark ("), the string item consists of all characters read between the first quotation mark and the second. If the first character of the string is not a quotation mark, the string is an unquoted string, and terminates with a comma, carriage return, or line feed (or after 255 characters have been read). If end of file is reached, the item is terminated.

Reference

Appendix A

IOCTL

Format

IOCTL [#]<file number>,<string>

Purpose

To send a control data string to a character device driver after the drive has been opened via the OPEN command.

Remarks

<file number> is the number of the file that has been opened.

<string> is a valid string expression containing the control data.

Examples

IOCTL commands are generally 2 to 3 characters optionally followed by an alphanumeric argument. An IOCTL command string may be up to 255 bytes long. Commands contained in the string must be separated by a semicolon (for example, "OO:SW132;GW").

If you have installed your own driver to replace LPT1 and that driver was able to set the page length (the number of lines to print on a page before issuing a form feed), then the following command can be used in an IOCTL statement to set or change the page length:

PLn	Where n is the new page length.

The following example opens the new LPT1 driver and sets the page length:

```
10 OPEN "LPT1:" FOR OUTPUT AS #1
20 IOCTL #1,"PL66"
```

The following example also opens LPT1, but with an initial page length of 56 lines:

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
20 IOCTL #1,"PL56"
```

Other user-definable IOCTL commands are possible, such as PTn (set print tabs every n spaces).

If the file that is represented by <file number> is not opened, a "Bad file number" error occurs.

If a device does not support IOCTL, an "Illegal function call" error occurs.

If the error is in control data, a "Device fault" error occurs.

Reference

IOCTL$

KEY

Format

```
KEY <key number>,<string expression>
KEY LIST
KEY ON
KEY OFF
KEY<key number>,CHR$(<shift>) + CHR$(<scan code>)
```

Purpose

To define and display the function key assignment text for function keys 1–10, or define keys 15–20 for event trapping.

Remarks

Refer to ON KEY for information on valid function keys. Use the ON KEY statement to use preassigned keys 11–14 for event trapping.

Defining Soft Keys. KEY <key number>, <string expression> assigns the string expression to the specified soft key (1–10). The defined string is input as a BASIC command when the assigned function key is pressed. If the string is longer than 15 characters, the first 15 characters are assigned.

Assigning a null string (with no characters) to a soft key disables the function key as a soft key.

If the value entered for <key number> is not in the range 1 to 10, an "Illegal function call" error occurs, and the key retains its previous string assignment.

The default functions of function keys 1–10 are as follows (<– indicates carriage return):

F1 LIST
F2 RUN<–
F3 LOAD"
F4 SAVE"
F5 CONT<–
F6 ,"LPT1:"<–
F7 TRON<–
F8 TROFF<–
F9 KEY
F10 SCREEN 0,0,0<–

When a soft key is assigned, INKEY$ returns one character of the soft key string each time it is used.

Example

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

This statement enables function key **F1** to perform a "CLS Return" sequence when it is pressed. This is how you change the functions of soft keys.

Displaying the Function Keys

KEY ON	Displays the key values on the 25th line (initial setting). Ten key values are displayed if the screen width is 80; five key values are displayed if the screen width is 40. In either case, the first six characters are displayed.
KEY OFF	Erases the soft key display from the 25th line. The function key assignment is not eliminated (even if the display is erased).
KEY LIST	Lists all ten soft key values on the screen. All 15 characters of each value are displayed.

Examples

```
30 KEY ON
```

This example displays the soft key display on the 25th line of the screen.

```
10 KEY OFF
```

This example removes the soft key display. The soft keys still work–they just aren't displayed on the screen.

```
40 KEY LIST
```

This example lists the values of each function key on the screen. The default values are as follows:

F1	LIST
F2	RUN<–
F3	LOAD"
F4	SAVE"
F5	CONT<–
F6	,"LPT1:"<–
F7	TRON<–
F8	TROFF<–
F9	KEY
F10	SCREEN 0,0,0<–

Event Trapping. Keys 15 to 20 allow you to define a sequence of characters to cause event trapping. For example, if you want your program to perform a specific action each time the key sequence **Ctrl R** is entered, you could define one of the keys (15–20) to trap that sequence and perform the action.

To define keys 15 through 20, use the following format:

```
KEY <key number>, CHR$(<shift>) + CHR$(<scan code>)
```

KEY <key number> is an unsigned integer in the range 15–20.

<shift> is a numeric value that corresponds to the hexadecimal value of the following keys:

Caps Lock	&H0 (not active)
	&H40 (active)
Num Lock	&H0 (not active)
	&H20 (active)
Alt	&H08
Ctrl	&H04
Left Shift	&H02
Right Shift	&H01
Either Shift	&H03

Combine multiple keys for <shift> by adding their respective hexadecimal values together. For example, <shift> set to &H08 + &H04 means the **Alt** and **Ctrl** keys are pressed simultaneously.

Since there are two **Shift** keys, &H01 and &H02, either value or the sum of the values, &H03, may be used for <shift>. The values used for <shift> must be hexadecimal numbers.

<scan code> is a hexadecimal or decimal number that identifies the key to be trapped. <scan code> is used with the key identified by <shift>. For example, to trap the sequence **Ctrl R** as key 15, use the following format:

```
KEY 15, CHR$(&H04) + CHR$(19)
```

The <shift> for the **Ctrl** key is &H04. The <scan code> for the letter **R** is 19 (decimal) or &H13 (hex). Hexadecimal values must be preceded by &H.

Refer to Appendix J for a list of keys and their scan codes.

Trapped keys are not entered into the keyboard buffer.

If you trap **Ctrl Break** and **Ctrl Alt Del**, you may not be able to stop your program without shutting the computer off. To trap these sequences, you should have a test in your program.

Ctrl PrtSc still activates the line printer, even if you attempt to trap it. Some scan codes are predefined and cannot be trapped. They are 59 through 68, 72, 77, and 80.

Example

```
300 KEY 15, CHR$(&H08 + &H04) + CHR$(19)
310 ON KEY (15) GOSUB 1200
320 KEY (15) ON
```

This statement traps the **Alt Ctrl R** sequence and performs the subroutine that starts on line 1200. The three statements KEY, KEY (<key number>), and ON KEY (<key number>) are all used to cause key trapping. The hexadecimal values for **Alt** and **Ctrl** are added together to show they are used simultaneously.

KEY ON/OFF/STOP

Format

```
KEY (<key number>) ON
KEY (<key number>) OFF
KEY (<key number>) STOP
```

Purpose

To enable, disable, or terminate the interrupt function of keys specified in an ON KEY statement.

Remarks

<key number> specifies a function key (1 through 20). ON KEY <key number> GOSUB <line number> sends program control to an interrupt subroutine (specified by GOSUB <line number>) when you press a specified <key number>). To enable, disable, or terminate the interrupt function of the specified key, use KEY ON, KEY OFF, and KEY STOP statements.

KEY <key number> ON enables the specified key to cause an interrupt.

KEY OFF disables interrupts. If the specified key is pressed, no interrupt occurs and BASIC doesn't record the fact that it was pressed.

KEY STOP, used anywhere in a program, terminates interrupts until another KEY ON statement is executed. When you press the specified key number, a KEY STOP statement causes BASIC to record interrupts, but not process them immediately. When a KEY ON statement is executed, BASIC processes all the recorded interrupts.

BASIC only allows the function keys to work during the execution of the main program. It automatically disables the interrupt function of <key number> as it enters the interrupt-handling subroutine, and automatically enables it as it leaves the subroutine via the RETURN statement. This prevents program control from getting caught in an endless loop within the interrupt-handling subroutine.

Interrupts only occur while BASIC is executing a program. If an ERROR interrupt occurs, all other interrupts (ERROR, PEN, COM, and KEY) are disabled.

INPUT$ and INKEY$ cannot be used to define the key numbers to cause interrupts. Use a separate ON KEY statement to specify each key number.

To assign the line number for return from the handling subroutine, use RETURN <line number>.

Example

```
10 ON KEY (1) GOSUB 190
20 ON KEY (2) GOSUB 200
30 ON KEY (3) GOSUB 210
40 ON KEY (4) GOSUB 220
50 FOR I=1 TO 4
60 KEY (I) ON
70 NEXT I
80 CLS
90 PRINT "Function key actions:"
100 PRINT "Function key F1 - Prints GOTO"
110 PRINT "Function key F2 - Prints PRINT"
120 PRINT "Function key F3 - Prints AUTO"
130 PRINT "Function key F4 - End of program"
140 PRINT
150 PRINT "Try pressing function key 1, 2, 3 or 4"
160 PRINT:PRINT:PRINT:
170 'This is an endless loop
180 GOTO 170
190 PRINT "GOTO": RETURN
200 PRINT "PRINT": RETURN
210 PRINT "AUTO": RETURN
220 PRINT "End of program": END
230 END
```

Reference

ON KEY, RETURN

KILL

Format

```
KILL "[<device name>:] <filename>"
```

Purpose

To delete a file from disk.

Remarks

KILL is used for all types of disk files: program files, random data files, and sequential data files.

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

KILL cannnot distinguish a file in another directory from one you have opened. You may get an unexpected "File already open" error in these circumstances.

If the filename has an extension, you must enter both the filename and the extension. Otherwise, "File not found" message occurs. (Include the device name if a file is not stored in the default drive.)

KILL allows path names to be included in device names.

Example

```
200 KILL "Data.Dat"
300 KILL "B:Data2.bas"
```

Reference

OPEN, Appendix A

LCOPY

Format

```
LCOPY
```

Purpose

To print the contents of the screen.

Remarks

The LCOPY statement sends the active page of the screen to a printer. It performs the same function as **Shift Prt Sc**.

LET

Format

[LET] <variable> = <expression>

Purpose

To assign the value of an expression to a variable.

Remarks

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

Examples

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
.
.
.
```

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
.
.
.
```

LINE

Format

LINE [(x1,y1)]-(x2,y2) [,[<color>] [,B[F]] [,<style>]]

Purpose

To draw straight lines and rectangles on the screen in graphic mode.

Remarks

The variables (x1,y1) and (x2,y2) are shown on the screen as absolute or relative coordinates.

For color displays, refer to the CIRCLE statement.

The LINE statement enables you to draw straight lines connecting the coordinates designated by (x1,y1) (x2,y2).

The simplest line format is:

LINE-(x2,y2)

This draws a line from the last point to the point (x2,y2) in the foreground attribute.

For example, if LINE-(100,100) is executed after LINE (0,0)-(10,10) a line is drawn connecting (0,0)-(10,10)-(100,100).

The coordinates for the final point can be written in relative form. In this case, the final point becomes the point defined by adding a specified offset to the coordinates of the first point. For example, the result of LINE (100,100)-STEP (20,-20) is the line segment (100,100)-(120,80).

The permissible range of the coordinates depends on the graphic mode. Refer to SCREEN.

If the coordinates specified are outside the screen range, a process called line clipping occurs. Lines are clipped at the boundaries of the viewing area so that only the points plotted within the screen are visible. Points outside the viewing area do not appear and will not wrap around.

For example, in high-resolution mode, any x coordinate greater than 639 is given the value 639 and any y coordinate greater than 399 is given the value 399. Negative coordinates are given the value 0. (When an x coordinate exceeds 319 in medium resolution mode, it goes as far as the next horizon line.)

Another argument to LINE is ,B (box) or ,BF (filled box). The syntax indicates that you can leave out the attribute argument and include the final argument as follows:

`LINE (0,0)-(100,100),,B`	Draw box in foreground or include it.
`LINE (0,0)-(200,200),2,BF`	Filled box attribute 2.

The ,B tells BASIC to draw a rectangle with points (x1,y1) and (x2,y2) as opposite corners. This avoids giving the following four LINE commands which perform the same function:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

,BF draws the same rectangle as ,B and fills in the interior points with a specified attribute.

<style> is a 16-bit integer mask used to put points on the screen (a technique called line styling). For example, when <style> is specified in binary, a one in the mask indicates that a dot is drawn in the corresponding position, and a zero indicates that a dot is not drawn in the corresponding position. When the line length is greater than 16 dots, the mask is used repeatedly. Note that a syntax error occurs if <style> is used with filled boxes (BF).

Example 1

```
10 SCREEN 3,,,,2: CLS:KEY OFF
20 FOR N=0 TO 255
30 LINE (N,N)-(319-N,399-N),N,B
40 NEXT
```

Example 2

```
100 SCREEN 1,0:COLOR 0,0
110 FOR I=1 TO 10
120 LINE (0,0)-(RND*319,RND*199), RND* 2+1
130 NEXT
140 FOR I=1 TO 10
150 LINE (319,199)-(RND*319,RND*199), RND*2+1
160 NEXT
170 GOTO 110
```

Example 3

```
10 SCREEN 1
20 LINE (0,0)-(160,100),3,,&HFF00
```

Example 3 draws a dashed line from the upper corner to the screen center.

Reference

CIRCLE, COLOR, SCREEN

LINE INPUT

Format

LINE INPUT[;] ["<prompt string>";] <string variable>

Purpose

To input an entire line (up to 254 characters) to a string variable, without using delimiters.

Remarks

The <prompt string> is a string literal entered at the terminal before it is accepted. A question mark is not displayed unless it is part of the string.

All input from the end of the prompt to the carriage return is assigned to <string variable>.

A carriage return signals the end of user input. However, if a line feed/carriage return sequence (in this order only) is encountered, both characters are echoed to the screen but the carriage return is ignored. The line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, a carriage return to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by pressing **Ctrl Break**. BASIC returns to command level and displays "Ok." Use CONT to resume execution at the LINE INPUT.

Reference

LINE INPUT#

LINE INPUT#

Format

```
LINE INPUT# <file number>,<string variable>
```

Purpose

To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks

<file number> is the number under which the file was opened.

<string variable> is the variable name to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It skips over the carriage return/line feed sequence. The next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is saved as part of a <string variable>.)

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

Example

```
10 OPEN "O",1,"List"
20 LINE INPUT "Customer information? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"List"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
Customer information? Linda Jones 234,4 Memphis
Linda Jones 234,4 Memphis
Ok
```

LIST

Format

LIST [<line number>][- [<line number>]] [,<filespec>]

Purpose

To list a program on the screen or another specified device.

Remarks

<line number> is a valid line number in the range 0 to 65529.

<filespec> is a valid string expression for file specification. (Refer to Chapter 2, "Naming and Accessing Files and Devices.")

If the optional device specification is omitted, the lines specified are listed on the screen and listing may be stopped at any time by pressing **Ctrl Num Lock**.

Listings directed to devices may not be interrupted. The list continues until the range is exhausted.

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

If only <line number> is specified, only that line is listed.

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

1. Only the first number is specified. That line and all higher numbered lines are listed.
2. Only the second number is specified. All lines from the beginning of the program to the specified line are listed.
3. Both numbers are specified. The range between the two numbers is listed.

If LIST is followed by a period, only the line at the present cursor position is displayed.

When files are listed on the disk, they are saved in ASCII format. Files saved in this manner can later be merged.

Examples

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.
LIST 150-	Lists lines from 150 to the end of the program.
LIST-1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.
LIST .	Lists the current statement.

LLIST

Format

LLIST [<line number>][-[<line number>]]

Purpose

To list all or part of the program currently in memory on the line printer.

Remarks

LLIST assumes an 80-character printer.

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

Examples

LLIST 150-	Lists all lines from 150 to the end of the program.
LLIST-1000	Lists all lines from the lowest number through 1000.
LLIST 150-1000	Lists lines 150 through 1000 inclusive.

Reference

LIST

LOAD

Format

```
LOAD "<filespec>" [,R]
```

Purpose

To load a file from a disk into memory.

Remarks

<filespec> is the file specification under which the file was saved. (With MS-DOS, include the default extension .BAS.)

LOAD closes all open files and deletes all variables and program lines currently in memory before it loads the program. If the R option is used, the program is run after it is loaded and all open data files are kept open. Thus, by using LOAD with the R option, information may be passed between programs or segments of the same program using disk data files.

Examples

```
LOAD "Strtrk",R
LOAD "b:strtrk",R
```

LOCATE

Format

LOCATE [<row>] [,[<col>] [,[<cursor>] [,[<start>] [,<stop>]]]]

Purpose

To move the cursor to a specified position on the screen. Optional parameters (<start> and <stop>) can be set to determine the size of the cursor.

Remarks

<row> is the screen line number (an unsigned integer in the range 1 to 25).

<col> is the screen column number (an unsigned integer in the range 1 to 40 or 1 to 80, depending on screen width).

<cursor> determines the display mode of the cursor:

0 = no display
1 = display

<start> and <stop> specify the size of the cursor. They are unsigned integers in the range 0 to 15. <Start> specifies the top scan line of the cursor and <stop> specifies the bottom scan line.

If you are using a color display, the bottom scan line is 7. In screen B (320x400 or 640x400), it is 15.

If you are using a monochrome display, the bottom scan line is 13.

If <start> is specified without <stop>, <stop> assumes the <start> value. If <stop> is specified without <start>, the cursor will not be displayed.

To redisplay the cursor, use LOCATE,,1.

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

Any values entered outside the allowable ranges cause an "Illegal function call" error. Previous values are retained.

Example

```
10 CLS
20 WIDTH 80
30 LOCATE 8,20: PRINT "Effect of <start> and <stop> on LOCATE
   statement"
40 FOR I=0 TO 8
50 FOR J=I TO 8
60 LOCATE 12,38,1,I,J: PRINT "START";I;
70 LOCATE 14,38,1,I,J: PRINT "STOP";J;
80 LOCATE 14,46,1,I,J
90 FOR N=1 to 800: NEXT N
100 NEXT J
120 NEXT I
130 LOCATE ,,,7,7
140 END
```

LPRINT and LPRINT USING

Format

```
LPRINT [<list of expressions>]
LPRINT USING <string exp>;<list of expressions>
```

Purpose

To print data at the line printer.

Remarks

These statements function like PRINT and PRINT USING, except the output goes to the line printer.

LPRINT assumes an 80-character printer.

Reference

PRINT and PRINT USING

LSET and RSET

Format

```
LSET <string variable> = <string expression>
RSET <string variable> = <string expression>
```

Purpose

To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks

If <string expression> requires fewer bytes than those fielded to <string variable>, LSET left-justifies the string in the field and RSET right-justifies the string. (Spaces are used to pad extra positions.) If the string is too long for the field, characters are dropped from the right.

Numeric values must be converted to strings before using LSET or RSET. Refer to MKI$, MKS$, and MKD$.

NOTE:

LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example,

```
110 N$ = SPACE$(20)
120 RSET A$ = N$
```

right-justifies string N$ in a 20-character field. This is useful for formatting printed output.

Examples

```
150 LSET A$ = MKS$ (AMT)
160 LSET D$ = DESC$ (3)
```

Reference

MKD$, MKS$, MKI$, Appendix B

MERGE

Format

MERGE "<filename>"

Purpose

To merge a disk file into the program currently in memory.

Remarks

<filename> is the name used when the file was saved. (With MS-DOS, include the default extension (.BAS). The file must have been saved in ASCII format. If it was not, a "Bad file mode" error occurs.

Any lines in the coment program that have the same line numbers as lines in the disk will be replaced. (Merging may be thought of as inserting program lines on a disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example

MERGE "Numbrs"

MID$

Format

MID$(<string exp1>,n[,m]) = <string exp2>

where n and m are integer expressions. <String exp1> and <string exp2> are string expressions.

Purpose

To replace a portion of one string with another string.

Remarks

<string exp1> and <string exp2> are string expressions. The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>.

The optional m specifies the number of characters from <string exp2> to be used in the replacement. If m is omitted, all of <string exp2> is used. Regardless of whether m is omitted or included, replacement of characters never goes beyond the original length of <string exp1>.

Example

```
10 A$ = "Kansas City, MO"
20 MID$(A$,14) = "KS"
30 PRINT A$
RUN
Kansas City, KS
```

MID$ also returns a substring of a given string. Refer to Chapter 4.

MKDIR

Format

MKDIR "<path>"

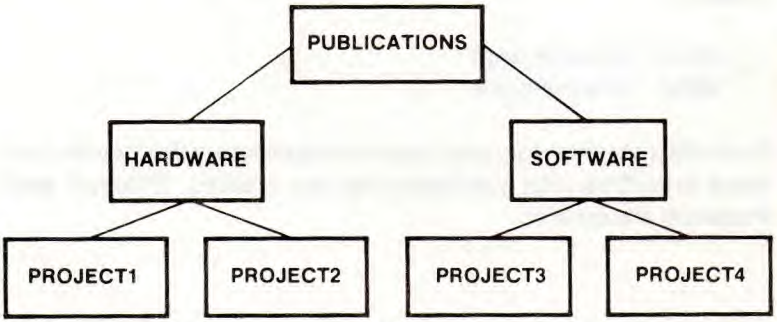
where <path> is:

[d:] [\] [<directory>\]. . .<directory>

Purpose

To create a directory on a diskette. <path> is a string expression that cannot exceed 63 characters.

The tree-structured directory with the root directory named Publications was created as follows:



Examples

```
MKDIR "Hardware"
```

From the root directory (Publications), a subdirectory (Hardware) is created.

```
MKDIR "Hardware\Project1"
MKDIR "Hardware\Project2"
```

From the root directory and under the subdirectory (Hardware), two more branches with subdirectories (Project1 and Project2) are created.

```
MKDIR "Software"
```

From the root directory another subdirectory (Software) is created.

```
MKDIR "Software\Project3"
MKDIR "Software\Project4"
```

From the root directory and under a subdirectory (Software), two more branches with subdirectories are created. (Project3 and Project4) are created.

The long way to create these two branches from the root directory is like this:

```
CHDIR "Software"
```

The subdirectory (Software) is the current directory.

```
MKDIR "Project3"
MKDIR "Project4"
```

The next level of subdirectories has been created.

For more information on tree-structured directories, refer to Chapter 2.

NAME

Format

```
NAME "<old filespec>" AS "<new filespec>"
```

Purpose

To change the name of a disk file.

Remarks

<old filespec> must be an existing filename or filespec.

<new filespec> must not exist. If this filespec exists, an error will result.

<old filespec> must be closed before renaming. Only files in the current directory may be renamed. Specifying a pathname will result in a "Bad filename" error message. After a NAME command, the old file is on the same disk, in the same area of disk space, with the new name.

Example

```
Ok
NAME "Accts" AS "Ledger"
Ok
```

The file that was formerly named Accts is now named Ledger.

NEW

Format

```
NEW
```

Purpose

To delete the program currently in memory and clear all variables.

Remarks

Enter NEW at command level to clear memory before entering a new program.

The NEW statement closes all open files.

ON COM

Format

ON COM (<circuit number>) GOSUB <line number>

Purpose

To define the starting line of the subroutine used when data arrives at the communication buffer.

Remarks

<circuit number> may be 1 or 2.

<line number> is the starting line of the communication handling routine.

To disable communication buffer interrupts, assign line 0 as the branching destination.

COM ON must be executed to activate ON COM.

ON COM only assigns the starting line of the subroutine. Branching to the subroutine after an interrupt only occurs when an outside factor (in this case, the arrival of data at the communication buffer) is present.

COM STOP disables interrupts from the communication buffer, but the arrival of data is recorded. Therefore, if data arrives after execution of COM STOP and COM ON is then executed, BASIC immediately goes to the line number assigned by COM ON.

COM STOP is executed automatically whenever an interrupt occurs (to prevent recalling the interrupt handling subroutine if another interrupt occurs during execution of the subroutine).

Unless COM OFF is executed during the handling subroutine, COM ON is executed automatically when RETURN is used at the end of the subroutine. This reenables interrupts from the communication buffer.

Interrupts occur only while BASIC is executing a program. If an ERROR interrupt occurs, all other interrupts (ERROR, PEN, COM, and KEY) are disabled.

To assign a destination for return from the handling subroutine, use RETURN or RETURN <line number>. RETURN <line number> must be used carefully because all other GOSUB, WHILE, or FOR statements active at the time of the subroutine trap remain active.

If possible, handle all data in a communication buffer at the same time. If characters are handled one at a time and the baud rate of the communication circuit is high, an increase in overhead time and a buffer overflow is possible.

Reference

COM, RETURN

ON ERROR GOTO

Format

ON ERROR GOTO <line number>

Purpose

To enable error trapping and specify the first line of the error handling subroutine.

Remarks

Error trapping causes all errors detected (including direct mode errors) to jump to the specified error handling subroutine.

If <line number> is not specified an "Undefined line" error results. To disable error trapping, execute ON ERROR GOTO 0. Subsequent errors display an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and display the error message for the error that caused the trap. If an error is encountered for which there is no recovery action, execute ON ERROR GOTO 0.

NOTE:

If an error occurs during execution of an error handling subroutine, the BASIC error message is displayed and program execution terminates. Error trapping does not occur within the error handling subroutine.

Example

```
10 ON ERROR GOTO 1000
```

Reference

ERROR

ON. . .GOSUB and ON. . .GOTO

Format

```
ON <expression> GOTO <list of line numbers>
ON <expression> GOSUB <list of line numbers>
```

Purpose

To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks

The value of <expression> determines which line number in the list will be used for branching. For example, if the value is 3, the third line number in the list is the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON. . .GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative (or greater than 255), an "Illegal function call" error occurs.

Example

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

In this example, if x-1 equals 1, BASIC goes to line 150. If x-1 equals 2, BASIC goes to line 300, and so on.

ON KEY

Format

```
ON KEY (<key number>) GOSUB <line number>
```

Purpose

To define the starting line of the subroutine used when a KEY interrupt occurs.

Remarks

<key number> is an unsigned integer in the range 1 to 20. <key number> represents specific function keys as follows:

1-10	Function keys 1 to 10
11	Cursor up key
12	Cursor down key
13	Cursor left key
14	Cursor right key
15-20	Keys defined by the programmer

To disable key interrupts, specify line 0 as the <line number>. KEY ON must be executed before ON KEY is executed.

ON KEY only defines the starting line of the subroutine. Branching to the subroutine after an interrupt only occurs when an outside factor (in this case, pressing a key) is present.

The ON KEY statement has been expanded to allow six additional user-defined key traps. This allows any key, the **Ctrl** key, the **Shift** key, or the **Alt** key to be trapped by the user.

Example

```
ON KEY (I) GOSUB <line number>
```

where I is an integer expression dependent on the number of function keys and direction keys.

There are ten function keys and four direction keys. Therefore, the user-defined keys are keys 15 through 20. User-defined keys are defined by the statement:

```
KEY I,CHR$(j) + CHR$ (k)
```

where I is the user-defined key number.

The user-defined keys are the first six keys following the function keys and direction keys.

Variable j is the mask for the latched keys: **Caps Lock, Num Lock, Alt, Ctrl**, left **Shift** key, and right **Shift** key. The appropriate bit in j must be set in order to trap a key that is used with **Ctrl** or **Alt**. Keyboard flag values in hex are:

&H40	**Caps Lock** is active.
&H20	**Num Lock** is active.
&H08	**Alt** key pressed.
&H04	**Ctrl** key pressed.
&H02	Left **Shift** key pressed.
&H01	Right **Shift** key pressed.
&H03	Either **Shift** key pressed.

Variable k identifies one of the keys on the keyboard.

The following rules apply to keys trapped by BASIC:

1. The **PrtSc** key is processed first. Defining the **PrtSc** key as a user-defined key trap will not keep it from sending characters to the line printer.
2. Function and cursor control keys are processed next. These keys cannot be redefined by the user.
3. The user-defined keys are processed last.
4. Any trapped key is not read by BASIC.

NOTE:

These rules apply to any key, including **Ctrl Break** or the **Ctrl Alt Del** reset sequence. This prevents you from accidentally interrupting a program with **Ctrl Break** or resetting the system.

Example

```
100 ON KEY (1) GOSUB 2000
110 KEY (1) ON
.
.
.
2000 'Subroutine called by function key 1
2100 TOTAL= SUBTOT1+SUBTOT2
2200 PRINT "The total is"; TOTAL
2300 RETURN
```

ON PLAY

Format

ON PLAY <n> GOSUB <line number>

Purpose

To play continuous background music during program execution.

Remarks

<n> is an integer expression that specifies the notes to be trapped. Values for <n> must be within the range 1 to 32, otherwise an "Illegal function call" error results. Be careful when choosing values for <n>. A value such as 32 causes so many event traps that little time is left to run the rest of the program.

<line> is the beginning line number of the PLAY trap routine. If the line number is 0, play trapping stops.

A PLAY ON statement enables the ON PLAY statement. If PLAY ON is used and the GOSUB <line number> specified in the PLAY statement is not 0, BASIC transfers control to <line> when notes remain in the background music buffer. This is called event trapping.

PLAY must be in the music background mode (PLAY "MB. . .") to execute an event trap. If PLAY is in the music foreground mode (PLAY "MF. . ."), an event trap is not executed.

If the music background buffer is empty when a PLAY ON statement is entered, PLAY is not executed.

If PLAY OFF is used, event trapping does not take place. Even if PLAY activity takes place, it is not stored.

If a PLAY STOP is used, event trapping does not take place, but PLAY activity is stored so that when PLAY ON is executed, an event trap takes place.

An automatic PLAY STOP runs when the trap occurs so that recursive traps can never take place. Unless a PLAY OFF was performed inside the trap routine, the RETURN statement from the trap routine automatically executes a PLAY ON.

BASIC must be running a program for event trapping to take place. If an ON ERROR statement causes an error trap, all trapping is automatically disabled (including ERROR, PEN, COM, and KEY).

To go back to the BASIC program at a fixed line number, use RETURN <line number>. RETURN must be used carefully, however, because any other GOSUB, WHILE, or FOR statement active at the time of the trap remains active.

Example

```
10 ON PLAY (8) GOSUB 1200                    'Set trap
20 PLAY ON                                   'Set music
30 REM Program to run in foreground
.
.
.
1200 REM Create melody for background music
.
.
1430 RETURN 30 'Go back to foreground program
```

If eight notes remain in the background music buffer, the trap executes GOSUB 1200. When the subroutine that starts at line 1200 ends, program control returns to line 30.

Reference

PLAY, RETURN

ON TIMER

Format

```
ON TIMER <n> GOSUB <line>
```

Purpose

To transfer control to a specific line number of a BASIC program after a defined period of time.

Remarks

<n> is a numeric expression ranging from 1 to 86,400 (1 second to 24 hours). If values outside this range are entered, an "Illegal function call" error message is displayed.

<line> is the beginning line number of the TIMER trap routine. TIMER trapping is disabled if line number 0 is specified.

A TIMER ON statement activates the ON TIMER statement. If TIMER ON is used and the line number specified in the ON TIMER statement is not 0 BASIC keeps track of time passed in seconds. After <n> seconds, BASIC performs a GOSUB to the line specified. The event trap occurs and BASIC starts counting from 0 again.

If TIMER OFF is used, trapping does not take place. Even if TIMER activity takes place, it is not remembered.

If a TIMER STOP statement is used, trapping does not take place, but TIMER activity is stored, so that when TIMER ON is used, an immediate trap occurs.

An automatic TIMER STOP is executed when the trap occurs so recursive traps cannot take place. Unless a TIMER OFF was performed inside the trap routine, a RETURN statement in the trap routine automatically executes a TIMER ON.

BASIC must be running a program for event trapping to take place. If an ON ERROR statement causes an error trap, all trapping is automatically disabled (including ERROR, PEN, COM, KEY, and PLAY).

RETURN <line number> makes it possible to go back to the BASIC program at a fixed line number. However, RETURN must be used carefully because any other GOSUB, WHILE, or FOR statement active at the time of the trap remains active.

ON TIMER is useful in programs that need an interval timer.

Example

```
10 ON TIMER (60) GOSUB 5000
20 TIMER ON
.
.
.
5000 SAVEROW=CSRLIN                Save the last row.
5010 SAVECOL=POS (0)               Save the last column.
5020 LOCATE 1,1                    Position cursor at top
5030 PRINT TIME$                   left of screen.
5040 LOCATE SAVEROW, SAVECOL       Return to saved position.
5050 RETURN
```

Reference

RETURN

OPEN

Format

```
OPEN <filespec> [FOR <file mode 1>]
AS[#] <file number> [LEN=<reclen>]

OPEN <path> [FOR mode] AS [#] <file number> [LEN=reclen]
OPEN "<file mode 2>", [#]<file number>,<filespec> [,<reclen>]
OPEN "<file mode 2>", [#]<file number>,<path> [,<reclen>]
```

Purpose

To open a file or device.

Remarks

In <file mode 1>, INPUT, OUTPUT, or APPEND are specified without quotation marks (""). If the file mode option is omitted, random access becomes the default mode.

If INPUT is specified for a file that does not exist, a "File not found" message is displayed.

If OUTPUT is specified for a file that does not exist, one is created.

<file mode 2> is a string expression whose first character is one of the following:

- O Specifies sequential output mode.
- I Specifies sequential input mode.
- R Specifies random input/output mode.

<file number> is an integer expression that can range from 1 to 255.

<filespec> specifies the name of the file and the name of the device to be opened. Refer to Chapter 2, "Files and Directories," for a review of file specifications.

<path> is a logical roadmap that traces the location of the file in the appropriate directories. Refer to Chapter 2 for a review of paths.

<reclen> is an integer expression in the range 1 through 32767. This value sets the record length for random files (refer to the FIELD statement). If omitted, the record length defaults to 128 bytes.

The OPEN statement allocates an input/output (I/O) buffer to the file and determines the mode of access to be used with the buffer. Once a file has been opened, <file number> can be used for I/O. The OPEN statement must be used before executing the following commands:

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

GET and PUT commands can be used only in files opened in random access mode. Only disk files and printers can be opened in this manner. When no device name is given, the disk drive currently in use is assumed.

Only one file can be opened for each file number.

For more details on communication files, refer to OPEN COM.

Examples

```
OPEN "B:Payroll" AS #5
```

The file (Payroll) located in drive B is opened for random access because <file mode 1> is omitted. File number 5 is associated with Payroll for subsequent referencing. The record length is omitted; therefore, a default record length of 128 bytes is assumed.

```
OPEN "Charges" AS 10 LEN=256
```

The file (Charges) located in drive A (default drive) is opened for random access and associated with file number 10 for subsequent referencing. (Use of the number sign (#) is optional.) The length of each record is 256 bytes.

```
OPEN "Publications\Software\Project4\Chapter3" FOR INPUT AS 12
```

The specified directory path (Publications\Software\Project4\Chapter3) is opened for input. Chapter3 is the filename. This path is linked with file number 12 for subsequent referencing. (The number sign (#) is optional.)

```
File$="Publications\Software\Project4\Chapter3"
OPEN File$ FOR OUTPUT AS 12
```

The first statement assigns this cumbersome path string expression to a string variable. In the OPEN statement, the string variable is used to represent the path.

```
OPEN "I", #2, "B:Taxes"
```

A file (B:Taxes) is opened for sequential input and is associated with file number 2 for subsequent referencing. File specification includes drive B and the filename (Taxes).

```
OPEN "R", 16, "Inventory"
```

A file (Inventory) is opened for random input and output and is associated with file number 16 for subsequent referencing. (The number sign (#) is optional.) Since the file is located in the default drive, the device name is not specified. The filename is Inventory. Since no record length is provided, the default of 128 bytes is assumed.

Reference

FIELD, OPEN COM

OPEN COM

Format

```
OPEN "COM: [<speed> [,<parity>[,<data>[,<stop>]]]] [,LF] [,RS]
          [,CS[n]] [,DS[n]] [,CD[n]] [PE]" AS [#]<file number>
          [LEN = number]
```

Purpose

To open a communications file.

Remarks

OPEN COM opens files for RS-232-C communications circuits. This command is similar to the OPEN command, except it applies to input and output of the RS-232-C communication circuit.

"COM: is a valid communication device. Valid devices are "COM1:, and "COM2:.

<speed> is a literal integer that specifies the transmit/receive baud rate. Valid speeds are 75, 110, 150, 300, 600, 1200, 1500, 2400, 4800, and 9600. The default for <speed> is 300.

<parity> is a 1-character literal that specifies the parity for transmit and receive, as follows:

S (Space)	Parity bit always transmitted and received as a space (0 bit).
O (Odd)	Odd transmit/receive parity checking.
E (Even)	Even transmit/receive parity checking (default).

N (None)	No transmit parity. No receive parity checking.
M (Mark)	Parity bit always transmitted and received as a mark (1 bit).

<data> is a literal integer that specifies the number of transmit/receive data bits. Valid values are 7 and 8.

<stop> is a literal integer that specifies the number of stop bits. Valid values are 1 and 2. If this variable is omitted, 110 bps transmits two stop bits. All others transmit one stop bit. (If a 4 or 5 is used for <data>, a 2-stop bit is interpreted as a 1.5-stop bit.)

The LF, RS, CS, DS, CD, and PE options affect the line signals. These options may appear in any order in the OPEN COM statement.

A definition of each option follows:

LF	Used when communications file data is being printed on a serial line printer. A line feed character (0A hex) is automatically appended to a carriage return character (0D hex). The LF option used in the OPEN COM statement also affects how data is read with INPUT# statements. Reading stops when a carriage return character is encountered. The line feed character is ignored by INPUT# and LINE INPUT# statements.
RS	Suppresses the request-to-send control character (1E hex). If the RS option is not used, the RTS action occurs automatically when the communications file is opened.

CS[n] Controls clear-to-send. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values are 0 to 65535, inclusive. The default value is 1000. If no [n] value is specified or the value equals 0, the line status is not checked.

Subsequent communications file I/O statements will be unsuccessful if the CS [n] option is not selected.

DS[n] Controls the "Data set ready" message. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values range from 0 to 65535 (inclusive). The default value is 1000. If no [n] value is specified or the value equals 0, the line status is not checked.

Subsequent communications file I/O statements will be unsuccessful if the DS[n] option is not selected.

CD[n] Controls the "Carrier detect" message. [n] specifies the number of milliseconds that pass before the host times out. Valid [n] values range from 0 to 65535 (inclusive). The default value is 0. If no value is given or the default is accepted, the line status is not checked.

The CD option is often referred to as the RLSD ("Received Line Signal Detect") message.

PE Activates parity checking. If the PE option is omitted, the default (no parity checking) is active. If a parity error is detected, a "Device I/O error" message is displayed. The high order bit is turned on for seven or fewer data bits. Framing and overrun errors will always turn on the high order bit and generate a "Device I/O error" message.

<number> (#) is the maximum number of bytes that can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

When **Ctrl Z** is received, the system treats it as the end of the file. When the file is closed, **Ctrl Z** is received in the circuit.

Tab changes position and CR is considered to be the line change sign at the end of a line. If LF is assigned, line feed is continuously output when CR is output.

<file number> is an integer expression returning a valid file number. The number is associated with the file as long as it is open and is used to refer other COM I/O statements to the file.

Missing parameters invoke the following defaults:

Speed	300 bps
Parity	Even
Bits	7

This command is similar to the OPEN command, except it applies to input and output of the RS-232-C communications circuit.

To transmit or receive numeric data, specify eight data bits.

A COM device may be opened to only one file number at a time.

A "Device timeout" error occurs if the RS-232-C asynchronous communications adapter is not installed correctly.

Reference

OPEN

OPTION BASE

Format

```
OPTION BASE n
```

where n is 1 or 0

Purpose

To declare the minimum value for array subscripts.

Remarks

The default base is 0.

If the statement,

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is 1.

A chained program may have an OPTION BASE statement if no arrays are passed. It will inherit the OPTION BASE value of the chaining program.

OUT

Format

```
OUT I,J
```

where I and J are integer expressions in the range 0 to 65535.

Purpose

To send a byte to a machine output port.

Remarks

Integer expression I is the port number, and integer expression J is the data to be transmitted.

OUT is complementary to the INP function.

Example

```
100 OUT 12345, 225
```

In assembly language, this is equivalent to:

```
MOV DX, 12345
MOV AL, 225
OUT DX, AL
```

PAINT

Format

PAINT (<x>,<y>) [,<paint> [,<border> [,<background>]]]

Purpose

To fill in a graphic figure or area of the screen with a specified color.

Remarks

<x> and <y> are the starting points of the PAINT statement. The <x> and <y> coordinates may be specified in absolute or relative form and must be within the area to be painted.

If the points (pixels) plotted are outside the screen limits, line clipping occurs. Pixels referenced outside the screen area are not drawn and do not wrap around.

<paint> and <border> are color codes. The figure to be painted is drawn in the <border> color and is filled in with the <paint> color.

<paint> and <border> are numeric expressions in the range 0 to 3 (depending on the graphic mode). Refer to the COLOR statement for a complete description of color values.

NOTE:

<paint> can also be a string expression, causing tiling to occur. Tiling is explained in this section.

There are only two colors in elementary and advanced screen A (high resolution mode). Therefore, <paint> and <border> colors should be the same. This whites out an area until white is encountered or blacks out an area until black is encountered.

PAINT must start on a non-border pixel. If the pixel specified by <x> and <y> already has the color <border>, PAINT has no effect.

If <paint> is omitted, the foreground color is used (color number 1 in high resolution and color number 3 in medium resolution).

If <border> is omitted, <paint> color is used. The <border> color should match the foreground color used to draw the graphic figure or area.

Any type of figure can be filled with PAINT. However, painting jagged edges or complex figures may require too much stack space, resulting in an "Out of memory" error. Use the CLEAR statement at the beginning of the program to increase the amount of stack space available.

Paint Tiling

Paint tiling allows you to design any type of figure or pattern within an 8-bit wide tile mask. The tile pattern is repeated uniformly over the entire screen.

To use paint tiling, <paint> must be a string expression in the following form:

```
CHR$(&Hnn) + CHR$(&Hnn) + CHR$(&Hnn)...
```

The two hexadecimal numbers in each CHR$ expression are converted to binary numbers and correspond to eight horizontal bits (four bits per n).

The length of the tile mask is determined by the number of tile bytes specified in the string. The string expression may contain up to 64 bytes. The structure of the string expression follows:

Tile Byte	Bit of Tile Byte								Coordinates
	7	6	5	4	3	2	1	0	x,y
0	x	x	x	x	x	x	x	x	0,0
1	x	x	x	x	x	x	x	x	0,1
2	x	x	x	x	x	x	x	x	0,2
									.
									.
									.
63	x	x	x	x	x	x	x	x	0,63

Point x increases from left to right and point y increases from top to bottom.

Tiling in High Resolution

In high resolution, there is only one bit per pixel. A point is plotted at every position in the bit mask that has a binary value of 1. For example, the expression CHR$(&H55) plots a point in every other bit of the 8-bit tile mask:

	Bit of Tile Byte								Tile Byte
	7	6	5	4	3	2	1	0	
CHR$(&H55)	0	1	0	1	0	1	0	1	0

The following example demonstrates painting a screen with x's in high resolution.

Example

```
10 SCREEN 2
20 PAINT (160,100), CHR$(&H81)+CHR$(&H42)+CHR$(&H24)+CHR$(&H18)+
   CHR$(&H18)+CHR$(&H24)+CHR$(&H42)+CHR$(&H81)
```

The length of the mask is 8, indexed 0 through 7. The pattern is as follows:

Expression	Bit of Tile Byte								Tile Byte
	7	6	5	4	3	2	1	0	
CHR$(&H81)	1	0	0	0	0	0	0	1	0
CHR$(&H42)	0	1	0	0	0	0	1	0	1
CHR$(&H24)	0	0	1	0	0	1	0	0	2
CHR$(&H18)	0	0	0	1	1	0	0	0	3
CHR$(&H18)	0	0	0	1	1	0	0	0	4
CHR$(&H24)	0	0	1	0	0	1	0	0	5
CHR$(&H42)	0	1	0	0	0	0	1	0	6
CHR$(&H81)	1	0	0	0	0	0	0	1	7

Tiling in Medium Resolution

In medium resolution, there are two bits per pixel. Therefore, only four pixels are specified in each tile byte. Every two bits specifies a color value for each of the four pixels to be plotted.

The following charts show the binary and hexadecimal values associated with each color.

Color Palette 0	Color Palette 1	Binary Color Number
Green	Cyan	01
Red	Magenta	10
Brown	White	11

Binary Pattern for Solid Line	Hexadecimal Pattern for Solid Line
01010101	&H55
10101010	&HAA
11111111	&HFF

Used in medium resolution, the expression CHR$(&H55) produces a solid green line in color palette 0 and a solid cyan line in palette 1:

CHR$(&H55) 0 1 0 1 0 1 0 1

The following example demonstrates plotting a pattern of boxes in medium resolution. If color palette 0 is used, the border color is green; if color palette 1 is used, the border color is cyan.

Example

```
10 SCREEN 1
20 PAINT (160,100), CHR$(&H55)+CHR$(&H41)+CHR$(&H41)+CHR$(&H41)+
   CHR$(&H41)+CHR$(&H41)+CHR$(&H41)+CHR$(&H55)
```

Expression	Bit of Tile Byte								Tile Byte
	7	6	5	4	3	2	1	0	
CHR$(&H55)	0	1	0	1	0	1	0	1	0
CHR$(&H41)	0	1	0	0	0	0	0	1	1
CHR$(&H41)	0	1	0	0	0	0	0	1	2
CHR$(&H41)	0	1	0	0	0	0	0	1	3
CHR$(&H41)	0	1	0	0	0	0	0	1	4
CHR$(&H41)	0	1	0	0	0	0	0	1	5
CHR$(&H41)	0	1	0	0	0	0	0	1	6
CHR$(&H55)	0	1	0	1	0	1	0	1	7

The starting point for plotting in paint tiling is determined by the following formula:

```
y mod tile-length
```

In the previous two examples, PAINT (at coordinates 160,100) begins by plotting byte 4. Eight bytes are specified in each string expression and y is specified as 100.

Use of the Background Attribute

Trying to tile over an area that is the same color as two consecutive lines in the tile pattern usually stops the paint process. Use <background> to continue.

An "Illegal function call" results if more than two lines in the tile pattern match the background color.

PALETTE

Format

PALETTE [<register number>,<color>]

Purpose

To assign a color to a color register. This statement is invalid in medium resolution screen B.

Remarks

<register number> is a numeric expression with a value in the range 0 to 15.

A register number is the same as the logical color code specified in a graphic statement or a COLOR statement.

<color> is a numeric expression with a value in the range 0 to 255 (for high resolution screen B) or 0 to 15 (for others). The same color may be assigned to two or more registers.

The relationship between the color codes and the actual colors for <color> follows.

1. When <color> is 0 to 255:

Color Code	Red	Green	Blue
0	000	000	00
1	000	000	01
2	000	000	10
3	000	001	11
4	000	001	01
.	.	.	.
.	.	.	.
.	.	.	.
32	001	000	00
.	.	.	.
.	.	.	.
.	.	.	.
255	111	111	11

   The red, green, and blue columns show contrasts. 000 is the darkest and 111 is the lightest. In the blue column, 00 is the darkest and 11 is the lightest.

2. When <color> is 0 to 15, the codes are the same as those for the COLOR statement.

   The effect of a PALETTE statement appears instantly on the screen.

   When BASIC is activated, the register contains the 16-color code the equals the register number.

   When the parameter is omitted, the old value is assumed.

Example

```
PALETTE 10, 2
```

Reference

COLOR

PALETTE USING

Format

```
PALETTE USING <array variable>
```

Purpose

To assign a color to color registers by integer-type array variables.

Remarks

A color code is assigned to <array variable> and written to color registers. This enables 16-color registers to change immediately.

The array must have at least 16 elements. To leave the contents of a specific register unchanged assign –1 to the array element corresponding to that register number (in advance).

Example

```
10 DIM A%(20)
20 A%(1)=2
30 A%(2)=3
.
.
.
100 PALETTE USING A%(10)
```

PLAY

Format

```
PLAY "<string expression>"
```

Purpose

To play a melody indicated by the character string.

Remarks

<string expression> is a string of single character commands enclosed in quotation marks. Spaces in <string expression> are ignored.

The single-character PLAY commands are:

A-G[#, + ,or–] Play the specified note. A # or + means sharp. A – means flat.

L <n> Sets the length of each note. Each note is expressed as Ln, where n ranges from 1 to 64, inclusive. Examples of notes follow:

Length	*Note*
L1	Whole note
L2	Half note
L3	One of a triplet of three half notes (1/3 of a 4-beat measure)
L4	Quarter note
L5	One of a quintuplet (1/5 of a measure)
L6	One of a quarter-note triplet
L64	Sixty-fourth note

MF (Music foreground.) Run music (PLAY statement) and SOUND in the foreground. Each subsequent note or sound will not start until the previous note or sound is finished. To escape the PLAY statement, press **Ctrl Break**.

MB (Music background.) Run music (PLAY statement) and SOUND in the background. Each note or sound is placed in a buffer allowing the BASIC program to continue while music plays in the background. Up to 32 notes (or rests) can be played in background at a time.

MN (Music normal.) Each note will play seven-eighths of the time determined by L (length).

ML (Music legato.) Each note will play the full period set by L (length).

MS (Music staccato.) Each note will play three-fourths of the time determined by L (length).

N<n> Play note <n>. The values for n may range from 0 to 84. In the seven possible octaves, there are 84 notes. 0 means rest.

><n> Rises to the next octave and plays note <n>, where n ranges from A through G, inclusive. Each time <n> is played, the octave rises until octave 6 is reached.

Example

```
PLAY ">a"
```

(Each time this statement is executed, the octave rises until 6 is reached and note "a" is played.)

<<n> Lowers to the next octave and plays note <n> when n ranges from A through G, inclusive. Each time <n> is played, the octave lowers until octave 0 is reached.

Example

```
PLAY "<a"
```

(Each time this statement is executed, the octave lowers until 0 is reached and note "a" is played.)

O <n> (Octave.) Sets the current octave. There are seven octaves (0 through 6). Each octave is on the scale from C to B.

P <n> (Pause.) May range from 1 to 64. Length is the same as Ln.

T <n> (Tempo) Sets the number of quarter notes (L4) in a minute. The values for n may range from 32 to 255. The default is 120.

. A dot or period after a note causes the note to play 3/2 times the value determined by L (length) times T (tempo). Multiple dots may appear after the note. The period is scaled accordingly. Dots may also appear after a pause (P) and scale the pause length.

Example

```
C. .
```

(In this case, c is 9/4 times as long as the length selected in Ln.)

X<string> (Execute substring.) The n in each command can be given either as a constant or a variable. If you use a string variable to define and store a melody, you can call it repeatedly by using x<string>. This enables you to play the same time with different tempos or octaves from another string. Use semicolons before and after an x<string> variable to separate it from the rest of the PLAY string.

Example

```
10 TUNE$ = "CEDFEGFDECD O2 B O3C"
20 PLAY "MN T150 L20 O3;XTUNE$;"
30 PLAY "ML T40 L64 O1;XTUNE$;"
40 END
```

To set character commands equal to a variable whose value changes within a program, use semicolons separating the variable from the rest of the PLAY string. The following example plays the same tune in five different octaves.

Example

```
10 FOR I = 1 to 5
20 PLAY "O = I;MS T100 L20
   CEDFEGFDECED O2 B O3 C"
30 NEXT I
```

POKE

Format

```
POKE I,J
```

where I and J are integer expressions.

Purpose

To write a byte into memory.

Remarks

Integer expression I is the address of the memory location (in the range 0 to 65535).

Integer expression J is the data to be written (in the range 0 to 255).

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. (Refer to Chapter 4.)

Both POKE and PEEK are useful for storing data, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example

```
10 POKE &H5A00,&HFF
```

Reference

PEEK

PRINT

Format

PRINT [<list of expressions>]

Purpose

To output data at the terminal.

Remarks

If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

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

If a comma or a semicolon terminates <list of expressions>, the next PRINT statement begins printing on the same line, spacing accordingly. If <list of expressions> terminates without a comma or semicolon, a carriage return occurs at the end of the line.

If the printed line is longer than the screen width, BASIC continues printing on the next line.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers that can be represented with six or fewer digits in the unscaled format as accurately as in the scaled format are printed in the unscaled format.

For example, 1E-7 is printed as .0000001. 1E-8(-7) is printed as 1E-08.

Double precision numbers that can be represented with 16 or fewer digits in the unscaled format as accurately as in the scaled format are printed in the unscaled format.

For example, 1D-15 is printed as .0000000000000001. 1D-16 is printed as 1D-16.

A question mark may be used instead of the word PRINT.

Examples

```
10 x=5
20 PRINT x+5, x-1, x*(-5), x^5
30 END
RUN
 10        4        -25        3125
Ok
```

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

```
LIST
10 INPUT x
20 PRINT x "squared is" x^2 "and";
30 PRINT x "cubed is" x^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 squared is 81 and 9 cubed is 729

? 21
 21 squared is 441 and 21 cubed is 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

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

In this example, the semicolons cause each value to be printed immediately after the preceding value. A number is always followed by a space and positive numbers are preceded by a space. In line 40, a question mark is used instead of the word PRINT.

PRINT USING

Format

```
PRINT USING "<string expression>";<list of expressions>
```

Purpose

To print strings or numbers using a specified format.

Remarks

<list of expressions> includes string expressions or numeric expressions to be printed, separated by semicolons.

<string expression> is a string literal (or variable) comprised of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers. String literals must be enclosed in quotation marks.

String Fields

When PRINT USING is used to print strings, the following characters may be used to format the string field:

!	Specifies that only the first character in the string is to be printed.
\n spaces\	Specifies that 2 + n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed. With one space, three characters are printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

Example

```
10 A$="Look":B$="Out"
20 PRINT USING "!";A$;B$
30 PRINT USING "\ \";A$;B$
40 PRINT USING "\    \";A$;B$;"!!"
RUN
LO
LookOut
Look  Out    !!
```

& Specifies a variable length string field. When the field is specified with "&", the string is printed exactly as it is input.

Example

```
10 A$="Look":B$="Out"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOut
```

Numeric Fields

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

#	Represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.
.	A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded.

Example

```
PRINT USING "##.##";.78
 0.78
PRINT USING "###.##";987.654
987.65

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

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

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

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

Examples

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95  +2.40  +55.60  -0.9

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

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The double asterisk also specifies positions for two more digits.

Example

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

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

Example

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

**$ Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

Example

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

, A comma is printed to the left of every third digit left of the decimal point. A comma at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

Example

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

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

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

Examples

```
PRINT USING "##.##^^^^";234.56
2.35E+02

PRINT USING "##.##^^^^-";-888888
88.89E+04-

PRINT USING "+##.##^^^^";123
+12.30E+01
```

The next character is printed as a literal character.

Example

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

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

%

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

Examples

```
PRINT USING "##.##";111.22
%111.22

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

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

PRINT# and PRINT# USING

Format

```
PRINT# <file number>, [USING "<string exp>";] <list of exps>
```

Purpose

To write data to a sequential disk file.

Remarks

<file number> is the number used when the file was opened.

<string exp> is comprised of formatting characters, as described in PRINT USING.

The expressions in <list of exps> are numeric or string expressions to be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk just as it would be displayed on the terminal with a PRINT statement. Delimit the data on the disk so it will be input correctly.

In <list of expressions>, delimit numeric expressions by using semicolons.

Example

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

(If commas are used as delimiters, the spaces inserted between print fields will also be written to disk.)

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

For example, let A$ = "Camera" and B$ = "93604-1". The statement

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

writes Camera93604-1 to the disk. Because there are no delimiters, the statement is not input as two separate strings. To correct the problem, insert delimiters as follows:

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

The following is written to disk:

```
Camera,93604-1
```

which can be read back as two string variables.

If the strings contain commas, semicolons, significant leading spaces, carriage returns, or line feeds, use CHR$(34) to enclose them in quotation marks.

If A$ + "Camera, Automatic" and B$ = "93604-1", the statement,

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

writes the following to disk:

```
Camera, Automatic93604-1
```

and the statement,

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

inputs "Camera" to A$ and "Automatic 93604-1" to B$.

To separate these strings properly on the disk, use CHR$(34) to write double quotation marks to the disk image. The statement,

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

writes the following to disk:

```
"Camera, Automatic""93604-1"
```

and the statement,

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

inputs "Camera, Automatic" to A$ and "93604-1" to B$.

The PRINT# statement may also be used with USING to control the format of the disk file.

Example

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

Reference

WRITE, Appendix A

PSET and PRESET

Format

PSET (<x>,<y>) [,<color>]
PRESET (<x>,<y>) [,<color>]

Purpose

To draw a dot at the assigned position on the screen.

Remarks

(<x>,<y>) are coordinates for setting a dot to be drawn. They may be absolute or relative.

<color> specifies the dot color.

The range of color values depends on the graphic mode used. (Refer to the CIRCLE statement.)

PRESET is the same as PSET. The only difference is that if no third parameter is specified in a PRESET statement, the background color 0 is selected.

If an out-of-range coordinate is specified for PSET or PRESET, no action is taken, nor does an error occur.

If <color> is a number greater than the permitted value, an "Illegal function call" error occurs.

Example

```
10 SCREEN 2 :CLS
20 FOR n=-100 TO 100
25 x=160+n
30 y=(n*n)/100
40 PSET(x,y)
50 NEXT
```

Reference

CIRCLE

PUT (Files)

Format

PUT [#]<file number> [,<record number>]

Purpose

To write a record from a random buffer to a random disk file.

Remarks

<file number> is the number under which the file was opened.

If <record number> is omitted, the next available record number (after the last PUT) is used. The largest possible record number is 16,777,215. The smallest record number is 1.

Reference

Appendix A

NOTE:

PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

PUT (For COM Files)

Format

```
PUT <file number>,<nbytes>
```

Purpose

To allow fixed-length I/O for COM:

Remarks

<file number> is an integer expression containing a valid file number.

<nbytes> is an integer expression containing the number of bytes to be transferred into or out of the file buffer.

PUT (Graphics)

Format

PUT (<x>, <y>), <array name> [,<operation>]

Purpose

To display graphic patterns in a specified position on the screen.

Remarks

(<x>, <y>) are coordinates of the upper left corner of the rectangular region on the screen.

<array name> is the name of the numerical array of the graphic pattern to be displayed on the screen.

<operation> specifies the operation to be performed with screen data when the graphic pattern appears on the screen. Operations include PSET, PRESET, XOR, OR, and AND. If <operation> is not specified, it is interpreted as XOR.

PUT displays array data on the screen.

The following operations may be selected for <operation>:

PSET	Displays the graphic pattern contained in the array as is (opposite of GET).
PRESET	Reverses the graphic pattern contained in the array and displays in (similar to a photographic negative).
OR	Displays the graphic pattern in the array, overlapping the data already displayed there.

XOR	Displays the result of the XOR operation on the screen data and the graphic pattern in the array. If points are already set at certain coordinates, data that corresponds to the graphic pattern in the array is reversed (NOT) at those points. If points are not set, data corresponding to the array is displayed on the screen. Specifying XOR displays patterns moving over a background.
AND	Displays the result of the graphic pattern contained in the array and data already displayed on the screen on a one-to-one basis.

If the graphic pattern is larger than the screen size, an "Illegal function call" error occurs.

Reference

GET

RANDOMIZE

Format

```
RANDOMIZE [<expression>]
RANDOMIZE TIMER
```

Purpose

To reseed the random number generator.

Remarks

<expression> may be any numeric formula specified as the random number seed. If <expression> is omitted, BASIC suspends program execution and asks for a value by displaying

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

before executing RANDOMIZE. If the same <expression> is used each time you run a program with RANDOMIZE, the same sequence of numbers is returned.

To change the sequence of random numbers each time the program is run, place a RANDOMIZE statement at the beginning of the program and change <expression> with each run.

To change the sequence of random numbers without a prompt, use the RANDOMIZE TIMER function. Each time you run the program, you'll get a different sequence.

Example

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
Ok
RUN
Random number seed (-32768 to 32767)? 3
 .2226007 .5941419 .2414202 .2013798 5.36178E-02
Ok
RUN
Random number seed (-32768 to 32767)? 4
 .628988 .765605 .551561 .775797 .7834911
Ok
RUN
Random number seed (-32768 to 32767)? 3
 .2226007 .5941419 .2414202 .2013798 5.361748E-02
Ok
```

READ

Format

READ <list of variables>

Purpose

To read values from a DATA statement and assign them to variables.

Remarks

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

A single READ statement may access one or more DATA statements (in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statements, an "Out of data" message is displayed.

If the number of variables specified is fewer than the number of elements in the DATA statements, subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

Examples

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

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

```
LIST
10 PRINT "City", "State", " Zip"
20 READ C$,S$,Z
30 DATA "Denver,",Colorado,80211
40 PRINT C$,S$,Z
Ok
RUN
City          State       Zip
Denver,       Colorado    80211
Ok
```

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

Reference

DATA, RESTORE

REM

Format

REM <remark>

Purpose

To insert explanatory remarks in a program.

Remarks

REM statements are not executed but are displayed exactly as entered when the program is listed.

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

Add remarks to the end of a line by preceding the remark with a single quotation mark instead of REM. A single quotation mark may be used instead of REM to form the entire explanatory remark.

NOTE:

Do not use REM in a DATA statement.

Examples

```
.
.
.
120 REM Calculate average velocity
130 FOR I=1 TO 20
140 SUM=SUM+V(I)
.
.
.
.
.
.
120 FOR I=1 TO 20'Calculate average velocity
130 SUM=SUM+V(I)
140 NEXT I
.
.
.
120 'Calculate average velocity
130 FOR I=1 TO 20
140 SUM=SUM+V(I)
```

Reference

GOTO, GOSUB

RENUM

Format

RENUM [[<new number>] [,[<old number>] [,<increment>]]]

Purpose

To renumber program lines.

Remarks

<new number> is the first line number to be used in the new sequence. The default is 10.

<old number> is the line in the current program where renumbering is to begin. The default is the first line of the program.

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

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

NOTE:

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

Examples

RENUM	Renumbers the entire program. The first new line number will be 10. Lines increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET

Format

```
RESET
```

Purpose

To close all disk files and delete all system buffers.

Remarks

If all open files are on disks, the RESET command is the same as CLOSE without specified file numbers.

Reference

CLOSE

RESTORE

Format

```
RESTORE [<line number>]
```

Purpose

To reread DATA statements from a specified line.

Remarks

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program.

If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57,68,79
50 PRINT A;B;C;D;E;F
RUN
57 68 79 57 68 79
.
.
.
```

Reference

CHAIN, DATA, READ

RESUME

Format

```
RESUME
RESUME 0
RESUME NEXT
RESUME <line number>
```

Purpose

To continue program execution after an error recovery procedure has been performed.

Remarks

Any of the four formats shown above may be used, depending upon where program execution is to resume:

RESUME or RESUME 0	Resumes at the statement that caused the error.
RESUME NEXT	Resumes at the statement immediately following the one that caused the error.
RESUME <line number>	Resumes at a specified line number.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message.

Example

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230)AND(ERL=90) THEN PRINT "Try again":RESUME 80
```

RETURN

Format

RETURN [<line number>]

Purpose

To return program control from the subroutine.

Remarks

<line number> is the line number to which you want to return. If you don't supply <line number>, program control automatically returns to the line number following the GOSUB statement that called the subroutine.

RETURN allows non-local returns from any subroutine, but it was especially designed for use with event-trapping routines. After an event-trapping routine, RETURN <line number> lets you return to the BASIC program at a point that eliminates the GOSUB entry the trap created.

RETURN must be used carefully since all other GOSUB, WHILE, or FOR statements active at the time of the trap remain active.

Example

```
10 OPEN "Empdata" FOR INPUT AS 1
.
.
.
100 GOSUB 1200 'Get employee data
110 WRITE #1, Name$, Addr$, Hiredate$
.
.
.
1200 INPUT "Employee Name"; Name$
1210 INPUT "Address";
1220 INPUT "Hiredate"; Hiredate$
1230 RETURN
```

Reference

GOSUB. . .RETURN

RMDIR

Format

RMDIR "<path>"

where <path> is:

[d:][\]<directory>\]. . .<directory>

Purpose

To delete a directory on a disk.

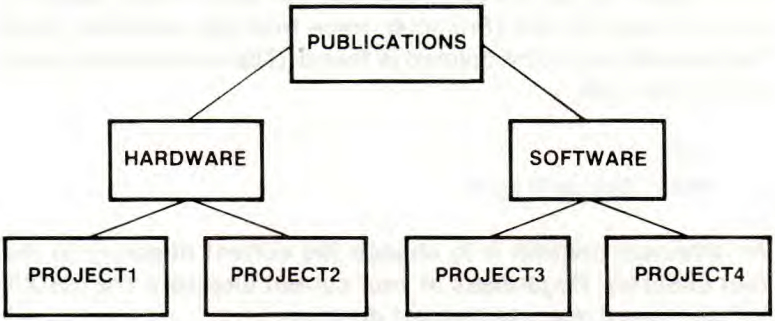
Remarks

<path> is a string expression that cannot exceed 63 characters.

Before deleting a directory, make sure it contains no subdirectories or files. Otherwise, an error message occurs.

For more information on tree-structured directories and file naming, refer to Chapter 2.

The following tree-structured directory with the root directory named Publications illustrates the RMDIR command.



Examples

```
RMDIR "Hardware\Project1"
```

From the root directory (Publications), the directory (Project1) is deleted. Trace from the top of the tree to the bottom until the subdirectory to be deleted is found. The subdirectory must not contain other files.

```
CHDIR "Software"
RMDIR "Project4"
```

To change the current directory, use the CHDIR command. To delete a subdirectory (Project4), trace from top to bottom until the subdirectory to be deleted is found. (The subdirectory must not contain files.)

```
CHDIR "\"
RMDIR "Software\Project4"
```

An alternate method is to change the current directory to the root directory. Regardless of your current directory, the CHDIR "\" command restores the root directory.

Trace from the top of the tree to the bottom until the desired subdirectory is found. (The subdirectory must not contain other files.)

NOTE:

When deleting subdirectories and files, do not remove the parent directory of the current directory. Always trace the path from top to bottom. Otherwise, an error message occurs. Attempting a KILL command to delete a directory also causes an error message.

Reference

CHDIR, MKDIR

RUN

Format

```
RUN [<line number>]
RUN "<filename>"[,R]
```

Purpose

To execute the program currently in memory or load a file from disk into memory.

Remarks

If <line number> is specified, program execution begins on that line. Otherwise, execution begins with the lowest line number. BASIC always returns to command level after a RUN is executed.

<filename> is the name that was used when the file was saved. (MS-DOS includes the default extension .BAS.)

RUN closes all open files and deletes the current contents of memory before loading the specified program. All data files remain open if the R option is used.

Examples

```
RUN
.
.
.
RUN "NEWFILE",R
.
.
.
```

Reference

Appendix A

SAVE

Format

```
SAVE "<filespec>"[,A]
SAVE "<filespec>"[,P]
```

Purpose

To save a program file on disk.

<filespec> is a quoted string that conforms to the operating system's requirements for filespecs. (By default, MS-DOS assumes .BAS if you do not include an extension.) If <filespec> already exists, the file is written over.

The program is saved to the file specified in <filespec>. If the device name is omitted, the disk drive in use is assumed.

If <filespec> is not specified, a "Path not found" error is displayed and the save is aborted.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some commands require that files be in ASCII format. For instance, the MERGE command requires an ASCII-format file and some operating system commands, such as LIST, may require an ASCII-format file.

Use the P option to protect the file by saving it in an encoded binary format.

If you do not specify A or P, BASIC defaults to the binary format.

Examples

```
SAVE "Com2",A
SAVE "Prog",P
```

NOTE:

When a protected file is later run (or loaded), any attempt to list or edit it causes an "Illegal function call" error. Files saved using the P option cannot be returned to their original format. Make a backup copy of your program before using the P option to save your program.

Reference

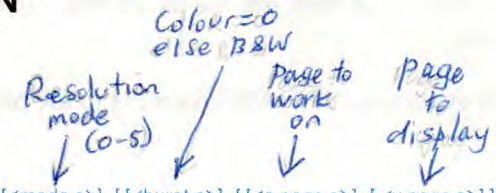
Appendix B

SCREEN

See page 2-39

Format

Screen A:




```
SCREEN [<mode a>] [,[<burst a>] [,[<a page a>] [,<v page a>]]]
```

Screen B:

(0-3)

```
SCREEN,,,, [<mode b>] [,[<burst b>] [,[<a page b> ][,<v page b>]]]
```

Superimposed (Screens A and B):

```
SCREEN [<mode a>], [<burst a>], [<a page a>], [<v page a>],
[<mode b>] [, [<burst b>] [, [<a page b>] [,<v page b>]]]
```

Purpose

To specify screen attributes for screen A, screen B, or superimposed screen (A and B).

Remarks

Options for <mode a> are:

Mode	Highest Page	Attribute
0	3(80) 7(40)	Elementary text (40 or 80 characters)
1	0	Elementary graphics (320x200)
2	0	Elementary graphics (640x200)
3	7	Advanced text (40 or 80 characters)
4	3	Advanced screen A (320x200)
5	3	Advanced screen A (640x200)

When a color display is connected, 1 through 5 are valid. Only 0 through 3 are valid for monochrome displays.

Options for <mode b> are:

Mode	Highest Page	Attribute
0	1	Advanced screen B (320x200)
1	1	Advanced screen B (640x200)
2	0	Advanced screen B (320x400)
3	0	Advanced screen B (640x400)

There is no text mode for screen B.

<burst a> and <burst b> are numeric expressions that enable or disable color for screens A and B, respectively.

For screen A, 0 disables color in the text modes (0 and 3). In medium resolution, elementary graphics, and advanced screen A graphics (1, 4 and 5), 0 enables color. In high resolution elementary graphics (2), only black and white are available, so this parameter is ineffective.

For screen B, 0 enables color.

<a page> is a numeric expression ranging from 0–7.

<a page a> or <a page b> selects the active page to be written to by output statements (LINE, COLOR, PRINT, etc.). Refer to Table 2–2 for the maximum number of pages available in each screen mode.

<v page a> or <v page b> specifies the visual page to be displayed on the screen. The visual page may be the same as the active page. If <v page> is omitted, BASIC assumes it is the same as the active page.

If parameters are omitted for screen A or screen B, all values (except v page) assume previously specified values. If both screens are used, they are superimposed. At least one attribute for each screen must be specified in the superimposed SCREEN statement.

Examples

```
10 SCREEN 0, 1, 0, 0
```

The color text mode is selected. <a page a> and <v page a> are both 0.

```
20 SCREEN ,, 1, 3
```

Mode and color burst remain unchanged. <a page a> is 1 and <v page a> is 3.

```
30 SCREEN 3,,,,2
```

Advanced text and 320x400 are specified for screen B. Superimposed mode is specified.

```
40 SCREEN,,,,3
```

640x400 is specified for screen B.

By manipulating active and visual pages as shown in the following example, you can display one page while you are creating another page. Then you can change the visual page to view the active page you created.

```
10 'Example of a page and v page in the screen statement
20 CLS:C1=224:C2=35
30 SCREEN 3,,0,1,0,,0,1 'viewing page 1, page 0 active
40 CIRCLE (160,100),50,C1,,,9
50 PAINT (160,100),C1
60 '
70 SCREEN 3,,1,0,0,,1,0 'viewing page 0, page 1 active
80 CIRCLE (160,100),50,C2,,,9
90 PAINT (160,100),C2
100 '
110 FOR N=1 TO 10
120 I=N MOD 2
130 SCREEN 3,,,,0,,I,I 'alternating view of page 0 and page 1
140 IF I=0 THEN LOCATE 13,38:IF I=0 THEN PRINT "Page 0"
150 IF I=1 THEN LOCATE 13,38:IF I=1 THEN PRINT "Page 1"
160 FOR J=1 TO 500: NEXT J
170 NEXT N
180 SCREEN 0,,0,0
190 END
```

SHELL

Format

SHELL [<command string>]

Purpose

To load and execute another program (.COM, .EXE, or .BAT). When the program finishes, control returns to the BASIC program at the statement following the SHELL statement. A program executed under control of BASIC is referred to as a child process.

Child processes (or children) are executed by SHELL loading and running a copy of COMMAND. By using COMMAND in this way, any parameters you may have are correctly parsed into the default FCB's, standard input and output may be redirected, and built-in commands such as DIR, PATH, and SORT may be executed.

Remarks

<command string> is a valid string expression containing the name of a program to run. Optionally, the string may contain command arguments.

The program name in <command string> may have an extension. If no extension is specified, COMMAND looks for a .COM file, then an .EXE file, and finally, a .BAT file. If none is found, SHELL issues a "File not found" error.

Text separated from the program name by at least one blank is processed by COMMAND as program parameters.

When a SHELL statement is executed, BASIC remains in memory while the child process is running. When the child process finishes, BASIC continues.

BASIC does not allow you to SHELL to another copy of BASIC. If you try to run BASIC as a child process, a "You cannot SHELL to BASIC" error occurs.

Discipline of Children. It is not possible for BASIC to totally protect itself from its children. When a SHELL statement is executed, many things may be going on. For example, files may be open and devices may be in use. The following guidelines will help prevent child processes from harming the BASIC environment.

Hardware. In general, it is recommended that the state of all hardware be preserved during a SHELL command. The implementation interface provides a way for performing this task. However, it may be necessary to request that you refrain from using certain devices within child processes that are executed using the BASIC SHELL command. Specific areas of concern are as follows:

a. Screen Device – Child processes might modify screen mode parameters. However, useful information may be displayed by a child process.

b. Interrupt Vectors – Save and restore interrupt vectors that the child intends to use. It seems reasonable to expect the child process to perform this task.

c. Other hardware – Many devices are placed in a specific state by BASIC. Such devices may include an interrupt controller, counter timer, DMA controller, I/O latch, and USART. These devices may be utilized by the child process without the user being aware of any limitations.

The File System. A child that alters any file opened in the BASIC parent may have disasterous results.

If it is necessary to update such files, they should be closed in the parent before executing a SHELL statement, then re-opened upon return to the BASIC parent.

Memory Management. Before BASIC SHELLs to COMMAND, it tries to free any memory that is not being used, except when BASIC is run with the /M: switch.

In this case, BASIC must assume that the user intended to load something in the top of BASIC's memory block. This prevents BASIC from "compressing the workspace" before doing the SHELL. For this reason SHELL may fail on an "Out of memory" error when using the /M: switch.

The preferred method is to load machine language subroutines before BASIC is run. This can be accomplished by placing "INT 27H" at the end of machine language subroutines. INT 27H allows machine language subroutines to exit to MS-DOS and stay resident. For example:

```
CSEG SEGMENT CODE

;Machine language subroutine

    RET ;Last instruction

START:
    INT 27H ;Terminate, stay resident
CSEG ENDS
    END START
```

Be sure to load these subroutines before BASIC by running them. The AUTOEXEC.BAT file is very useful for this.

A child process should never terminate and stay resident. Doing so may not leave BASIC enough room to expand its workspace to the original size. If BASIC cannot restore the workspace, all files are closed, the error message "Can't continue after SHELL" is printed, and BASIC exists to MS-DOS.

SOUND

Format

SOUND <frequency>,<duration>

Purpose

To generate sound through the speaker.

Remarks

<frequency> specifies the frequency in Hertz (a valid numeric expression containing an unsigned integer in the range 37 to 32767).

<duration> specifies the duration in clock ticks (a valid numeric expression containing an unsigned integer in the range 0 to 65535). Clock ticks occur 18.2 times per second.

There is no break in program execution when sound is generated with the SOUND statement.

If <duration> is 0, any current SOUND statement is turned off.

If <duration> is not 0, the next SOUND statement is not executed until the sound generated by the previous SOUND statement ends.

In order to obtain an interval in which no sound is produced, specify 32767 as <frequency>.

Example

```
10 A=37
20 B=.1
30 SOUND A,B
40 A=A+10
50 IF A=2167 THEN 70
60 GOTO 30
70 SOUND A,B
80 A=A-10
90 IF A=37 THEN GOTO 40
100 GOTO 70
```

STOP

Format

```
STOP
```

Purpose

To terminate program execution and return to command level.

Remarks

STOP statements may be used anywhere in a program. When a STOP is encountered, the following message is displayed:

```
Break in line nnnn
```

Unlike the END statement, STOP does not close files.

BASIC always returns to command level after a STOP is executed. Resume program execution by using a CONT command.

Example

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
30.7693
Ok
CONT
115.9
Ok
```

SWAP

Format

```
SWAP <variable>,<variable>
```

Purpose

To exchange the values of two variables.

Remarks

Any type variable may be exchanged (integer, single precision, double precision, or string), but the two variables exchanged must be the same type or a "Type mismatch" error occurs.

Example

```
LIST
10 A$="one " :B$="all " :C$="for "
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
one for all
all for one
Ok
```

SYSTEM

Format

```
SYSTEM
```

Purpose

To exit BASIC and return to MS-DOS.

Remarks

When SYSTEM is executed, all files are closed before returning to MS-DOS. All programs and data in memory are also lost, so use this statement carefully.

TRON/TROFF

Format

```
TRON
TROFF
```

Purpose

To trace the execution of program statements.

Remarks

The TRON statement (in direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets.

The trace flag is disabled by using TROFF (or when a NEW command is executed).

Example

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K + 10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

VIEW

Format

VIEW [[SCREEN] [(<x1>,<y1>)-(<x2>,<y2>)
[,[<color>] [,[<boundary>]]]]]

Purpose

Defines viewports (sections of the viewing surface) onto which window contents are mapped. VIEW allows you to change the size of the viewport, causing objects to appear smaller or larger (scaling). VIEW can also create multiple viewports, but only one can be active at a time.

Remarks

Coordinates (<x1>,<y1>) and (<x2>,<y2>) define the boundaries of the viewport. (<x1>,<y1>) is the upper left corner. (<x2>,<y2>) is the lower right corner.

VIEW sorts the *x* and *y* pairs by placing the smallest values first.

Example

```
VIEW (50,50)-(10,10)
```

is sorted to

```
VIEW (10,10)-(50,50)
```

Example

```
VIEW (50,100)-(100,50)
```

is sorted to

```
VIEW (50,50)-(100,100)
```

The only restriction on the possible pairings is that <x1> cannot equal <x2> and <y1> cannot equal <y2>. All other pairs are valid (if the coordinates are within the actual limits of the screen). An "Illegal function call" error occurs if the coordinates define a viewport larger than the viewing surface.

<color> is an optional variable that allows you to fill the specified viewport with color. <color> can range from 0 to 3, depending on the resolution.

In medium resolution mode, the color is defined by the COLOR statement from the current palette. The default colors are 0 for background and 3 for foreground.

In high resolution mode, only black and white are available. The defaults for this mode are 0 (black) for background and 1 (white) for foreground.

<boundary> is also optional. It allows you to draw a boundary line around the viewport (space permitting). If <boundary> is omitted, no boundary line is drawn. A color ranging from 0 to 3 (as described above) is also used for <boundary>.

If SCREEN is included, all pixels (points) plotted are absolute. That is, although only pixels within the viewport limits are visible, points plotted may be inside or outside the screen limits.

Example

```
5 REM Example of VIEW SCREEN
10 KEY OFF: CLS: SCREEN 1,0
20 WINDOW SCREEN (320,0)-(0,200)
30 VIEW SCREEN (20,20)-(220,120),,1
40 CIRCLE (50,50),50,1,,,1
50 END
```

If SCREEN is omitted, all pixels plotted are relative to the viewport as defined by the x and y coordinates. Values <x1> and <y1> are added to the coordinates before the point is plotted on the screen.

Example

```
5 REM Example of VIEW
10 KEY OFF: CLS: SCREEN 1,0
20 WINDOW SCREEN (320,0)-(0,200)
30 VIEW (20,20)-(220,120)
40 CIRCLE (100,60),30,1,,,1
```

If VIEW is used with no options, the viewport is defined as the entire viewing surface. In medium resolution, the viewing surface is VIEW (0,0)–(319,199). In high resolution, the viewing surface is (0,0)–(639,199). In medium resolution, you view SCREEN 1; in high resolution, you view SCREEN 2.

Multiple viewports may be specified, but only one viewport is active at a time. RUN and SCREEN disable the viewports.

There are two variations used to clear the screen and send the cursor to home position. To clear the entire screen, use VIEW to disable the viewports; then, use CLS to clear the screen.

Using the CLS statement without first using the VIEW statement clears only the current viewport. Using CLS with viewports does not home the cursor. Use **Ctrl Home** to home the cursor and clear the screen.

VIEW also scales (changes the size of the viewport). Objects appear large with a large viewport and small with a small viewport. Scaling with VIEW is similar to zooming with the WINDOW statement.

Examples

```
5 'Using VIEW to do SCALING
10 KEY OFF: CLS: SCREEN 1,0
20 WINDOW SCREEN (320,0)-(0,200)
30 'Draw the picture:
40 GOSUB 80: FOR I=1 TO 1000:NEXT I:CLS
50 'Make the picture smaller
60 VIEW (40,40)-(280,160),,1:GOSUB 80
70 END
80 'Subroutine to draw the picture
90 CIRCLE (120,100),50,1,,,1
100 RETURN
```

```
5 REM Example of Three Viewports
10 KEY OFF: CLS: SCREEN 1,0: VIEW
20 LOCATE 8,2:PRINT "First Viewport"
30 VIEW (30,20)-(90,80),,1
40 CIRCLE (60,50),20,1,,,1
50 LOCATE 14,6:PRINT "Second Viewport"
60 VIEW (110,40)-(210,160),,1
70 CIRCLE (60,50),40,1,,,1
80 LOCATE 18,18: PRINT "Third Viewport"
90 VIEW (230,120)-(290,180),,1
100 CIRCLE (60,50),20,1,,,1
110 END
```


Reference

COLOR, WINDOW, PMAP

VIEW PRINT

Format

```
VIEW PRINT [<top line> TO <bottom line>]
```

Purpose

Sets the boundaries of the text window in which information is written to the screen and scrolling occurs.

Remarks

<top line> is an integer line number that specifies the uppermost line in the text window. <bottom line> specifies the last line in the text window. If no parameters are entered, the whole screen is included in the text window.

By default, the text window comprises 24 lines. When you use the /G option in the BASIC command line, the text window comprises 25 lines (see Chapter 2).

Statements and functions that operate within the text window include CLS, LOCATE, and the SCREEN function.

Example

The following VIEW PRINT statement defines an 11-line text window from line 10 to line 20.

```
VIEW PRINT 10 to 20
```

WAIT

Format

WAIT <port number>, I [,J]

where I and J are integer expressions.

Purpose

To suspend program execution while monitoring the status of an input port. (The WAIT statement waits until certain conditions are satisfied at the machine input port.)

Remarks

The data read at the port is compared with integer expression J. The result of this comparison is then compared with integer expression I.

BASIC allows program execution to continue past the WAIT statement only if the data read at the port is not equal to integer expression J, and if integer expression I is greater than zero.

NOTE:

It is possible to enter an infinite loop with the WAIT statement, in which case it is necessary to manually restart the terminal.

Example

```
100 WAIT 32,2
```

WHILE. . .WEND

Format

```
WHILE <expression>
.
.
.
[<loop statements>]
.
.
.
WEND
```

Purpose

To execute a series of statements in a loop as long as a given condition is true.

Remarks

If <expression> is not 0 (true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error message. An unmatched WEND statement causes a "WEND without WHILE" message.

Example

```
10 REM Initialize array
20 DIM A(50)
30 FOR I=50 TO 1 STEP -1
40 A(51-I)=I
50 NEXT I
60 REM Print the unsorted array
70 FOR J=1 TO 50
80 PRINT A(J);
90 NEXT J
100 PRINT
110 REM Bubble sort array A$
120 FLIPS=1 'Force one pass through loop
130 WHILE FLIPS
140 FLIPS=0
150 FOR I=1 TO 49
160 IF A$(I)>A$(I+1) THEN SWAP A$(I),A$(I+1):FLIPS=1
170 NEXT I
180 WEND
190 REM Print sorted array
200 FOR J=1 TO 50
210 PRINT A(J);
220 NEXT J
230 END
```

WIDTH

Format

```
WIDTH <size>
WIDTH <file number>, <size>
WIDTH "<dev>", <size>
```

Purpose

To set the width in number of characters for the screen or line printer.

Remarks

<size> is a valid numeric expression containing an integer in the range 0 through 255 to specify the new width.

<file number> is a valid numeric expression containing an integer in the range 1 through 255.

<dev> is a valid string expression containing the device identifier. Valid device identifiers are: SCRN:, LPT1:, LPT2:, LPT3:, COM1:, and COM2:.

BASIC automatically adds a carriage return and line feed when the specified width is reached.

Depending upon the device specified, the following statements set the screen width:

```
WIDTH <size>
WIDTH "SCRN:", <size>
```

Only a 40- or 80-column width is allowed.

NOTE:

Changing the screen width clears the screen.

If WIDTH 40 (or WIDTH "SCRN:",40) is used in the 80-character mode, the screen accepts only 40 characters. Use the SCREEN statement to change from a 40-character to an 80-character width screen.

If the screen is in medium resolution graphic mode (SCREEN 1 or 4), WIDTH 80 forces the screen into high resolution graphic mode (SCREEN 2 or 5).

If the screen is in high resolution graphic mode (SCREEN 2 or 5), WIDTH 40 forces the screen into medium resolution graphic mode (SCREEN 1 or 4).

In superimposed mode, only an 80-character width screen is available for screen A.

WIDTH "<dev>",<size> stores the new width value without actually changing the current width setting. A subsequent OPEN, LIST, or LLIST statement uses this value to set the new width.

When the format WIDTH <file number>,<size> is used:

<file number> resets the currently open file to the output width.

WIDTH cannot be executed for the keyboard (KYBD:).

A file may be associated with LPT1:, LPT2:, LPT3:, COM1:, and COM2:.

No error message appears even if the specified print width is larger than can be accepted by the printer in use.

Specifying WIDTH 255 for the line printer disables line folding. This has the effect of infinite width.

Reference

SCREEN

WINDOW

Format

WINDOW [[SCREEN](<x1>,<y1>)-(<x2>,<y2>)]

Purpose

To redefine the coordinates of the screen, allowing you to draw objects in space so the logical screen can exceed the boundaries of the physical screen.

Remarks

(<x1>,<y1>) and (<x2>,<y2>) are programmer-defined, single precision, floating point numbers called world coordinates. They define the world coordinate space that will be mapped into the physical coordinate space.

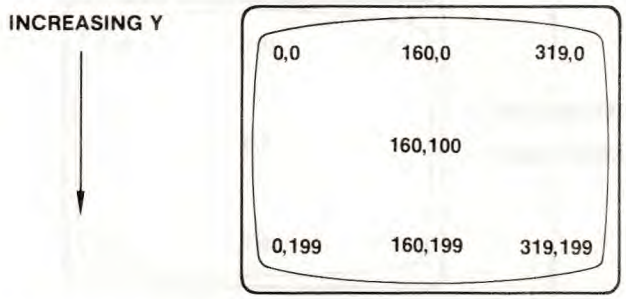
The physical coordinate space is defined by the VIEW statement. The rectangular region in the world coordinate space (defined by the world coordinate pairs), is called a window.

The WINDOW statement redefines the screen to be more dense or less dense than the physical screen. The VIEW statement can reduce the window and make it more dense. WINDOW and VIEW are similar in that they both can change the density of the picture, but the VIEW statement can make the window smaller than the physical window.

BASIC converts the world coordinate pairs to physical coordinates for screen display.

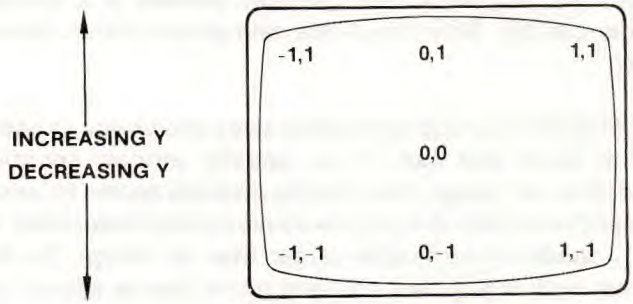
Standard coordinates in the physical coordinate system vary depending upon the screen mode.

Standard Coordinates:



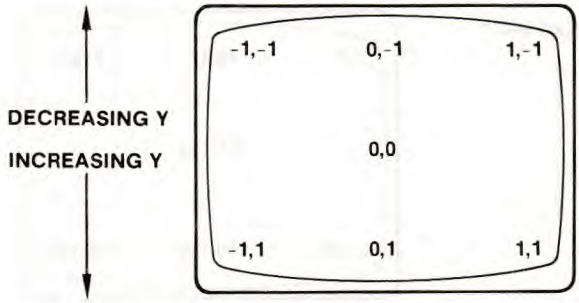
[SCREEN] is optional. If it is omitted, the screen is viewed in true Cartesian coordinates. (The y coordinate is inverted so that (<x1>,<y1>) is the lower left coordinate and (<x2>,<y2>) is the upper right coordinate.)

WINDOW Without SCREEN Attribute:



If [SCREEN] is included, the coordinates are not inverted. (<x1>,<y1>) is the upper left coordinate and (<x2>,<y2>) is the lower right coordinate.

WINDOW With SCREEN Attribute:



WINDOW sorts the coordinate pairs by placing the smallest values first. Value <x1> cannot equal <x2> and value <y1> cannot equal <y2>, but all other possible pairings are valid.

The WINDOW statement uses a process called line clipping. With the clipping process, pixels (points) referenced outside a coordinate range are invisible to the viewing area. An object that lies partially within and partially outside of a coordinate range is cut off. Only the pixels referenced within the range appear.

Specifying small and large window sizes allows you to use WINDOW to zoom and pan. If you specify window coordinates smaller than an image, the clipping process zooms in, allowing only a portion of the image to be displayed and magnified. If you specify window coordinates larger than an image, the image pans out until it is quite small and blank spaces appear on the sides of the screen.

Using the RUN, SCREEN, and WINDOW commands with no options disables any window coordinates and returns the screen to physical coordinates.

The PMAP function is used to learn the relationship between the physical screen and the world.

Examples

```
5 REM Using Window to Show Clipping
10 KEY OFF: CLS: SCREEN 2
20 WINDOW (-10,70)-(10,10)
30 CIRCLE (3,1),4,1 'Picture clipping shown here
40 WINDOW (-50,-50)-(50,50)
50 CIRCLE (3,1),4,1 'Picture is very small
60 END
```

```
10 'Example of window and window screen statements
20 '
30 CLS : KEY OFF
40 SCREEN 3,,,,2
50 'First, draw a figure using default screen values
60 WINDOW SCREEN (0,0)-(320,400)
70 LOCATE 1,30 : PRINT "Window screen (0,0)-(320,400)"
80 X1=160 : Y1=200
90 CIRCLE (X1,Y1),20,160
100 FOR I=1 TO 3000 : NEXT I
110 LOCATE 1,30 : PRINT STRING$(35,&H20)
120 'Now make the window smaller than original
130 LOCATE 1,20 : PRINT "Window gets smaller - figure looks larger"
140 FOR N=12 TO 120 STEP 8
150 WINDOW (N,N)-(320-N,400-N)
160 CIRCLE (X1,Y1),20,180
170 CIRCLE (X1,Y1),20,0
180 NEXT N
190 CIRCLE (X1,Y1),20,180
200 FOR I=1 TO 2000 : NEXT I
210 LOCATE 1,20 : PRINT STRING$(35,&H20)
220 END
```

WRITE

Format

```
WRITE [<list of expressions>]
```

Purpose

To output data at the terminal.

Remarks

If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric or string expressions. They must be separated by commas.

When the items are output, each item will be separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement.

Example

```
10 A=80:B=90:C$="That's all"
20 WRITE A,B,C$
RUN
80,90,"That's all"
Ok
```

WRITE#

Format

```
WRITE# <file number>,<list of expressions>
```

Purpose

To write data to a sequential file.

Remarks

<file number> is the number under which the file was opened in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary to specify delimiters in the list. A carriage return/line feed sequence follows the last item in the list.

Example

If A$ = "Camera" and B$ = "93604-1", the statement

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

writes the following image to disk:

```
"Camera","93604-1"
```

A subsequent INPUT# statement, such as

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

inputs "Camera" to A$ and "93604-1" to B$.

Reference

PRINT$

Chapter 4. BASIC Functions and Variables

Chapter 4. BASIC Functions and Variables

Intrinsic functions provided by this version of BASIC are presented in this chapter. The format of the descriptions is the same as that in Chapter 3.

Functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In this chapter, arguments are abbreviated as follows:

x and y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is specified where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

NOTE:

Functions return integer and single-precision results unless you use the /D option in the BASIC command line. When you use the /D option, the following functions return double-precision results: ATN, COS, EXP, LOG, SIN, SQR, and TAN.

ABS

Format

```
ABS(<x>)
```

Purpose

To return the absolute value of the expression <x>.

Example

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

ASC

Format

```
ASC(<X$>)
```

Purpose

To return a numeric value that is the ASCII code of the first character of the string <X$>. If <X$> is null, an "Illegal function call" error is returned. (Refer to Appendix H for ASCII codes.)

Example

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

Reference

CHR$

ATN

Format

```
ATN(<x>)
```

Purpose

To return the arctangent of <x> in radians. The result is in the range -pi/2 to pi/2. <X> may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example

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

CDBL

Format

```
CDBL(<x>)
```

Purpose

To convert <x> to a double precision number.

Example

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

CHR$

Format

```
CHR$(<I>)
```

Purpose

To return a string whose one element has ASCII code <I>. CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character (CHR$(7)) could be sent as a preface to an error message or a form feed (CHR$(12)) could be sent to clear the screen and return the cursor to home position.

ASCII codes are listed in Appendix H.

Example

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

Reference

ASC, Appendix H

CINT

Format

```
CINT(<x>)
```

Purpose

To convert <x> to an integer by rounding the fractional portion. If <x> is not in the range −32768 to 32767, an "Overflow" error occurs.

Example

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

Reference

CDBL, CSNG

COS

Format

COS(<x>)

Purpose

To return the cosine of <x> in radians. The calculation of COS(<x>) is performed in single precision.

Example

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

CSNG

Format

```
CSNG(<x>)
```

Purpose

To convert <x> to a single precision number.

Example

```
10 A# = 975.342184#
20 PRINT A#; CSNG(A#)
RUN
 975.342184 975.3422
Ok
```

Reference

CINT, CDBL

CSRLIN

Format

```
CSRLIN
```

Purpose

To return the current line (or row) position of the cursor.

Remarks

A value in the range 1 to 25 is returned.

X = POS(0) returns the column location of the cursor. (To set the cursor position, use the LOCATE statement.)

Reference

LOCATE, POS, SCREEN

CVI, CVS, CVD

Format

```
CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)
```

Purpose

To convert string values to numeric values. Numeric values read from a random disk file must be converted from strings to numbers.

CVI converts a 2-byte string to an integer.

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

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

Example

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

Reference

MKI$, MKS$, MKD$, Appendix A

DATE$

Format

DATE$ = <string exprssion>	(Sets the current date.)
<string expression> = DATE$	(Returns the current date.)

Purpose

To set or return the current date.

Remarks

The date is in 10-letter form (mm-dd-yyyy) where m is month, d is day, and y is year. The date is set by MS-DOS before BASIC is loaded.

Format the date as follows:

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

The year range is 1980 to 2077. If the month or day has only one digit, put a zero (0) in front of it. If the year is specified with only two digits, 19yy or 20yy is set automatically.

ENVIRON$

Format

V$ = ENVIRON$ (<parm>)
V$ = ENVIRON$ (<nth parm>)

Purpose

To return the specified environment string from the environment string table.

Remarks

<parm> is a valid string expression containing the parameter to search for.

<nth parm> is an integer expression returning a value in the range 1 to 255.

If a string argument is used, ENVIRON$ returns a string containing the text following <parm> from the environment string table.

If <parm> is not found, or no text follows <parm>, then a null string is returned.

ENVIRON$ distinguishes between uppercase and lowercase letters. Typically, <parm> consists of all uppercase letters, such as "PATH".

If a numeric argument is used, ENVIRON$ returns a string containing the <nth parm> from the environment string table including the <parm> text.

If there is no <nth parm>, then a null string is returned.

Example

Unless changed by the MS-DOS PATH command, the initial environment string is empty. For example, suppose that you entered the following ENVIRON statement:

```
ENVIRON "PATH=C:\SALES;C:\ACCOUNTING"
```

The environment string table would contain:

```
PATH=C:\SALES;C:\ACCOUNTING
```

If this is the case, the statement:

```
PRINT ENVIRON$("PATH")
```

will print the string:

```
C:\SALES;C:\ACCOUNTING
```

and

```
PRINT ENVIRON$(1)
```

will print the string:

```
PATH=C:\SALES;C:\ACCOUNTING
```

The following program saves the environment string table in an array so that it may be modified for a child process. After the child process completes, the environment is restored.

```
10 DIM ENV.TBL$(10) 'Assume no more than 10 parms
20 N.PARMS= 1 'Initial number of parms
30 WHILE LEN (ENVIRON$ (N.PARMS) >0
40 ENV.TBL$(N.PARMS) = ENVIRON$(N.PARMS)
50 N.PARMS = N.PARMS + 1
60 WEND
70 N.PARMS = N.PARMS-1 'Ajust to correct number
80 'Now store new environment
90 ENVIRON "MYCHILD.PARM1 = SORT BY NAME"
100 ENVIRON "MYCHILD.PARM2 = LIST BY NAME"
.
.
.
1000 SHELL "MYCHILD" 'Runs MYCHILD.EXE
1010 FOR I= 1 TO N.PARMS
1020 ENVIRON ENV.TBL$(I) 'Restore parms
1030 NEXT I
.
.
.
```

Errors that may occur include "Illegal function call" if <nth parm> is out of range, "Type mismatch" if <parm> is not a string, and "String too long" if the string is longer than 255 characters.

Reference

ENVIRON

EOF

Format

EOF <file number>

Purpose

To determine whether the file specified in <file number> has ended.

Remarks

<file number> is specified by <filespec> in the OPEN statement.

The EOF function returns -1 (true) or 0 (false) in response to whether the end of a sequential or communications file has been reached.

If true (-1), the output queue is empty.

Use EOF to test for the end of file and avoid "Input past end" errors.

ERDEV/ERDEV$

Format

```
V  = ERDEV
V$ = ERDEV$
```

Purpose

To return information concerning a device error.

ERDEV returns the error code.

ERDEV$ identifies the device that caused the error. If a character device caused the error, ERDEV$ returns the device name. If a block device caused the error, ERDEV$ returns the drive letter followed by a colon.

Remarks

ERDEV and ERDEV$ are read-only system variables.

ERDEV is set by the interrupt X'24' handler when an error within MS-DOS is detected.

ERDEV returns the INT 24 error code in the lower 8 bits. The upper 8 bits contains the word attribute bits (bits 15 to 13) from the device header block.

If the error occurred on a character device, ERDEV$ returns the 8-byte character device name. If the error did not occur on a character device, ERDEV$ contains the two-character block device name (A:, B:, C:, etc.).

Example

User installed device driver MYLPT2 caused a "Printer out of paper" error via INT 24.

ERDEV contains the error number 9 in the lower 8 bits and the device header word attributes in the upper 8 bits.

ERDEV$ contains the string "MYPLT2".

ERR and ERL

Format

```
u =ERR
u =ERL
```

Purpose

To return the error code and line number associated with an error.

Remarks

When an error handling subroutine is entered, ERR contains the error code for the error and ERL contains the number of the line in which the error was detected.

ERR and ERL are usually used in IF. . .THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL contains 65535. To find out whether an error occurred in a direct mode statement, use IF 65535 = ERL THEN. . . Otherwise, use:

```
IF ERR = <error code> THEN. . .
IF ERL = <line number> THEN. . .
```

If the line number is not to the right of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET statement.

BASIC error codes are listed in Appendix F.

Reference

IF . . .THEN, LET

EXP

Format

```
EXP(<x>)
```

Purpose

To return the value of e to the power of <x>. <x> must be less than or equal to 88.02969. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is the result, and program execution continues.

Example

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

FIX

Format

```
FIX(<x>)
```

Purpose

To return the truncated integer portion of <x>. FIX(<x>) is the equivalent of SGN(x*INT(ABS(x))). FIX is similar to INT, except it does not return the next lower number for negative <x>.

Examples

```
PRINT  FIX(58.75)
 58
Ok

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

Reference

INT

FRE

Format

```
FRE(0)
FRE(<X$>)
```

Purpose

To return the number of bytes in memory (not being used by BASIC).

Remarks

The arguments used in FRE are dummy arguments.

FRE forces a garbage collection before returning the number of free bytes. The computer often provides more space than necessary for variables, arrays, and files. FRE locates all the unused space in memory, and makes it available for use.

BASIC will not initiate garbage collection until all free memory has been used. Therefore, periodically using FRE results in shorter delays for garbage collection.

Garbage collection may take 1 to 1½ minutes.

Example

```
10 X$ = "Number of free bytes."
20 PRINT FRE(X$)
RUN
 62170
Ok
```

HEX$

Format

```
HEX$(<x>)
```

Purpose

To return a string that represents the hexadecimal value of the decimal argument. <x> is rounded to an integer before HEX$(<x>) is evaluated.

Example

```
10 INPUT x
20 A$ = HEX$(x)
30 PRINT x "Decimal is " A$ " hexadecimal"
RUN
? 32
 32 Decimal is 20 hexadecimal
Ok
```

Reference

OCT$

INKEY$

Format

```
INKEY$
```

Purpose

To read a single character from the keyboard and return a zero, one, or two-character string. A null string is returned if no character is pending at the terminal. A one-character string contains the actual character read from the terminal. A two-character string indicates a special extended code that is returned for special keys, such as **Home, End,** and **Insert**. Refer to the end of Appendix H for a complete list of those keys that return a two-character string.

The result of INKEY$ must be assigned to a string variable.

No characters are echoed and all characters are passed through to the program (except **Ctrl Break**, which terminates the program).

Example

```
10 'Input response timer
20 RANDOMIZE TIMER
30 SECS = 138
40 CLS
50 FOR I=1 TO 750 *3: NEXT I
60 A = INT(RND*20+5)
70 N=0
80 LOCATE 1,1: PRINT A
90 SOUND 400,3
100 N = N + 1
110 X$ = INKEY$
120 IF LEN(X$) = 0 THEN 100
130 IF N<A * SECS THEN PRINT "Too early"
140 IF N >(A+1) * SECS THEN PRINT "Too late"
150 IF N>A *SECS -.5 AND N < (A + 1)* SECS -.5 THEN PRINT
    "Right on!"
160 FOR I=1 TO 750 *3 :NEXT I
170 GOTO 40
```

INP

Format

```
INP(<I>)
```

Purpose

To return the byte read from port <I>. <I> must be in the range 0 to 65535. The INP function is the complement of the OUT statement.

Example

```
100 A = INP(5432)
```

In assembly language, this is equivalent to:

```
MOV DX,5432
IN AL,DX
```

Reference

OUT

INPUT$

Format

```
INPUT$(<x>[,[#]<y>])
```

Purpose

To return a string of <x> characters read from the terminal or from file number <y>.

If the terminal is used for input, no characters are echoed and all control characters are passed through (except **Ctrl Break**, which cancels INPUT$).

Examples

```
5 'List the contents of sequential file in hexadecimal
10 OPEN "I",1, "DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
.
.
.
100 PRINT "Type P to proceed or S to stop"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
```

INSTR

Format

```
INSTR([<I>,]<X$>,<Y$>)
```

Purpose

To search for the first string <Y$> in <X$> and return the position at which the match is found.

Optional offset <I> sets the position for starting the search. (<I> must be in the range 1 to 255.) If <I> is greater than LEN <X$>, <X$> is null. If <Y$> cannot be found, INSTR returns to 0. If <Y$> is null, INSTR returns I or 1.

<X$> and <Y$> may be string variables, string expressions, or string literals.

Examples

```
10 X$ = "abcdeb"
20 Y$ = "b"
30 PRINT INSTR(X$,Y$); INSTR(4,X$,Y$)
RUN
 2 6
Ok
```

NOTE:

If 0 is specified for I, an error message is returned.

INT

Format

INT(<x>)

Purpose

To return the largest integer less than or equal to <x>.

Examples

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

Reference

CINT, FIX

IOCTL$

Format

```
V$ = IOCTL$([#]<file number>)
```

Purpose

To read a control data string from a character device driver that is opened.

Remarks

<file number> is the number of a file that has been opened.

The IOCTL$ function is generally used to get acknowledgment that an IOCTL statement succeeded or failed. It is also used to get device information after an IOCTL statement is issued.

Example

```
10 OPEN "\DEV\FOO" AS #1
20 IOCTL #1,"RAW" 'Tell device that data is "raw"
30 IF IOCTL$(1) ="0" THEN CLOSE 1
```

If the character device driver FOO returns a false value from the raw data mode IOCTL$ request, the file is closed.

If the file that is represented by <file number> is not opened, a "Bad file number" error occurs.

If the device does not support IOCTL, an "Illegal function call" error occurs.

Reference

IOCTL

LEFT$

Format

```
LEFT$(<X$>,<I>)
```

Purpose

To return a string comprised of the leftmost <I> characters of <X$>. <I> must be in the range 0 to 255. If <I> is greater than LEN(<X$>), the entire string (<X$>) is returned. If <I> equals 0, the null string (length zero) is returned.

Example

```
10 A$ = "BASIC86"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
Ok
```

Reference

MID$, RIGHT$

LEN

Format

```
LEN(<X$>)
```

Purpose

To return the number of characters in <X$>. Non-displayed characters and blanks are counted.

Example

```
10 X$ = "Portland Oregon"
20 PRINT LEN(X$)
RUN
 15
Ok
```

LOC

Format

LOC(<file number>)

Purpose

To return the present location in the file.

Remarks

With random disk files, LOC returns the record number just read or written.

With sequential files, LOC returns the number of sectors (in 128-byte blocks) read from or written to the file since it was opened. If a sequential file is opened in the input mode, BASIC reads the first sector and the LOC function value is set at 1.

Numeric values given in random access and sequential files are in 128-byte blocks.

For a communications file, LOC returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: option in the BASIC initialization command). If there are more than 255 characters in the queue, LOC returns 255. A string is limited to 255 characters. This limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 255 characters remain in the queue, LOC returns the actual count.

LOF

Format

LOF(<file number>)

Purpose

To return the number of bytes allocated to the file.

Remarks

<file number> specifies a currently open file.

When the disk file is specified, the actual size of the file is returned in byte units. In BASIC, this file is in units of 128 bytes; therefore, the value is an integer multiple of 128.

When the communication file is specified, LOF returns the amount of free space in the input buffer. That is:

/C:<size>-LOC(<file number>)

The buffer size is ordinarily 256, but it is possible to change it with the /C: option.

For random files, LOF returns the actual record position in the file. This is the last record in the file plus 1.

For sequential files, or when a file is opened for APPEND or OUTPUT, LOF returns the size of the file in bytes.

Example

```
100 OPEN "O",#1,"Test.dat"
110 INPUT A$
120 PRINT #1,A$
130 CLOSE #1
140 OPEN "I",#2,"Test.dat"
150 PRINT LOF(2)
160 CLOSE #2
170 END
```

LOG

Format

```
LOG(<x>)
```

Purpose

To return the natural logarithm of <x>. (<x> must be greater than zero.)

Example

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

LPOS

Format

```
LPOS(<x>)
```

Purpose

To return the current position of the line printer printhead within the line printer buffer. LPOS does not necessarily give the physical position of the printhead.

<x> is a numeric expression that specifies a printer device name:

0,1	LPT1:
2	LPT2:
3	LPT3:

Example

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

MID$

Format

```
MID$(<X$>,<I>[,<J>])
```

Purpose

To return a string of length <J> characters from <X$> (beginning with <I>). <I> and <J> must be in the range 1 to 255.

If <J> is omitted (or if there are fewer than <J> characters to the right of <I>), all rightmost characters beginning with <I> are returned.

If <I> is greater than LEN(<X$>), MID$ returns a null string.

Example

```
10 A$ = "Good"
20 B$ = "morning evening afternoon"
30 PRINT A$;MID$(B$,8,8)
RUN
Good evening
Ok
```

Reference

LEFT$, RIGHT$

NOTE:

If 0 is specified for I, an error message is returned.

MKI$, MKS$, MKD$

Format

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Purpose

To convert numeric values to string values. Any numeric value placed in a random file buffer with an LSET or RSET statement must be converted to a string.

MKI$ converts an integer to a 2-byte string.

MKS$ converts a single precision number to a 4-byte string.

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

Example

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$, 20 AS N$
110 LSET D$=MKS$(AMT)
120 LSET N$=A$
130 PUT #1
```

Reference

CVD, CVI, CVS, Appendix A

OCT$

Format

```
OCT$(<x>)
```

Purpose

To return a string that represents the octal value of the decimal argument. <x> is rounded to an integer before OCT$(<x>) is evaluated.

Example

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

Reference

HEX$

PEEK

Format

PEEK(<I>)

Purpose

To return the byte (decimal integer in the range 0 to 255) read from memory location <I>. (<I> must be in the range 0 to 65536.)

PEEK is complementary to the POKE statement.

Example

```
A = PEEK(&H5A00)
```

Reference

POKE

PLAY

Format

```
PLAY(<n>)
```

Purpose

To report the number of notes currently stored in the music background buffer.

Remarks

PLAY (<n>) reports the maximum number of notes stored in the buffer when music background (MB) mode is used. The maximum value reported is 32. (A maximum of 32 notes can be held in the buffer.)

PLAY returns zero when the program is operating in music foreground mode.

Example

```
100 IF PLAY(0) = 10 GOTO 50
```

PMAP

Format

PMAP(<x>,<n>)

Purpose

To map the physical coordinates to world coordinates or to map world coordinates to physical coordinates.

Remarks

PMAP is used to map coordinates between the world coordinate system specified in the WINDOW statement and the physical coordinate system specified in the VIEW statement.

<x> is the coordinate of the point being mapped.

<n> is a value in the range 0 to 3.

0 Maps world coordinate<x> to physical coordinate <x>.

1 Maps world coordinate <y> to physical coordinate <y>.

2 Maps physical coordinate <x> to world coordinate <x>.

3 Maps physical coordinate <y> to world coordinate <y>.

Given the world coordinate <x>, PMAP returns the physical coordinate <x>. If the world coordinate <y> is specified, the physical coordinate <y> is returned.

Reference

VIEW, WINDOW

POINT

Format

POINT(<x>,<y>)
POINT(<n>)

Purpose

To provide information about a point (pixel) on the screen. POINT (<x>, <y>) returns the value of a specified pixel on the screen.

POINT <n> returns the value of the current x or y graphic coordinate.

Remarks

<x> and <y> are coordinates of the specified pixel. Coordinates must be in absolute form, indicating the physical position of the pixel on the screen.

The color code of the dot with coordinates assigned by (<x>, <y>) is returned as a numeric value.

Valid color values returned in medium resolution are 0, 1, 2, and 3. In high resolution, valid values are 0 and 1. The value -1 is returned if the point specified is outside the range of the screen.

The following chart shows the attribute values for each type of graphic mode:

Graphic Mode			Attribute Value
Elementary		320x200	0-3
		640x200	0,1
Advanced	Screen A	320x200	0-3
		640x200	0,1
	Screen B	320x200	0-255
		640x200	0-15
		320x400	0-255
		640x400	0-15

POINT <n> returns the value of the current x or y graphic coordinate. The value of <n> can range from 0 to 3:

0 Returns the current physical x coordinate.

1 Returns the current physical y coordinate.

2 Depends upon the WINDOW statement. If WINDOW is active, 2 returns the current world x coordinate. If WINDOW is not active, 2 returns the current physical x coordinate.

3 If WINDOW is active, 3 returns the current world y coordinate. If WINDOW is not active, 3 returns the current physical y coordinate.

Examples

```
10 SCREEN 3,,,,3:CLS:KEY OFF
20 ICOLOR=2
30 FOR x=40 TO 120:PSET (x,50), ICOLOR:NEXT x
40 FOR y=20 TO 80:PSET (80,y), ICOLOR:NEXT y
50 z=POINT (80,50)
60 PRINT:PRINT TAB(13); "x__,__y"
70 PRINT TAB(12); "(80,50)"
80 FOR I=1 TO 7:PRINT:NEXT I
90 PRINT "The color code at (80,50) is ";z:"."
```

This example illustrates color values returned by the POINT function.

```
10 KEY OFF:SCREEN 1,0:CLS
20 PRINT "POINT with WINDOW inactive"
30 GOSUB 110
40 WINDOW (0,0)-(319,199)
50 PRINT "POINT with WINDOW active"
60 GOSUB 110
70 PRINT "POINT with WINDOW and SCREEN active"
80 WINDOW SCREEN (0,0)-(319,199)
90 GOSUB 110
100 END
110 PSET (5,15)
120 FOR N=0 TO 3
130 PRINT POINT (N)
140 NEXT N
150 RETURN
```

This example demonstrates the effect of the WINDOW statement on the values returned by the POINT function.

Reference

WINDOW

POS

Format

```
POS(<x>)
```

Purpose

To return the current column position of the cursor. The left-most position is 1.

<x> is a dummy argument.

Example

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

Reference

LPOS

RIGHT$

Format

```
RIGHT$(<X$>,<I>)
```

Purpose

To return the rightmost <I> characters of string <X$>. If <I> is equal to LEN(<X$>), <X$> is returned. If I equals 0, the null string (length zero) is returned.

Example

```
10 A$="Disk BASIC"
20 PRINT RIGHT$(A$,5)
RUN
BASIC
Ok
```

Reference

LEFT$, MID$

RND

Format

```
RND[(<x>)]
```

Purpose

To return a random number between 0 and 1. The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. (Refer to RANDOMIZE.) However, if <x> is less than 0, the same sequence for any specified <x> is generated.

If <x> is greater than 0 or <x> is omitted, the next random number in the sequence is generated. If <x> is equal to 0, the last number generated is repeated.

Examples

```
10 FOR I=1 TO 5
20 PRINT INT (RND (15)*100)
30 NEXT
RUN
 12  65  86  72  79
Ok
```

Reference

RANDOMIZE

SCREEN

Format

<x> =SCREEN(<row>,<col>[,<z>])

Purpose

To return the ASCII code (0-255) for the character at a specified row and column of the screen.

Remarks

<x> is a numeric variable receiving the ordinal returned.

<row> is a valid numeric expression containing an unsigned integer in the range 1 to 25.

<col> is a valid numeric expression containing an unsigned integer in the range 1 to 40 or 1 to 80 (depending on the screen width).

<z> is a valid numeric expression containing a Boolean result. If <z> is specified and is not 0, the color attribute for the character is returned. If <z> is 0 or is not specified, the ASCII code corresponding to the character is returned. If <z> is specified, <x> means:

(x MOD16) = foreground color

(((x-foreground)/16)MOD 128) = background color

SGN

Format

```
SGN(<x>)
```

Purpose

To return the value of <x> that is greater than, less than, or equal to zero

If <x> is greater than 0, SGN(x) returns 1. If <x> is equal to 0, SGN(x) returns 0. If <x> is less than 0, SGN(x) returns -1.

Example

```
ON SGN(x)+2 GOTO 100,200,300
```

This example branches to 100 if <x> is negative, 200 if <x> is 0, or 300 if <x> is positive.

SIN

Format

```
SIN(<x>)
```

Purpose

To return the sine of <x> in radians. SIN(<x>) is calculated in single precision as follows:

```
COS(x) = SIN(x + 3.14159/2)
```

Example

```
PRINT SIN(1.5)
 .9974951
Ok
```

Reference

COS

SPACE$

Format

```
SPACE$(<x>)
```

Purpose

To return a string of spaces the length of <x>. (<x> is rounded to an integer and must be in the range 0 to 255.)

Example

```
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
1
 2
  3
   4
    5
Ok
```

Reference

SPC

SPC

Format

```
SPC(<I>)
```

Purpose

To display blanks on the terminal, SPC can only be used with PRINT and LPRINT statements.

<I> must be in the range 0 to 255. A semicolon is assumed to follow the SPC(<I>) command.

Example

```
PRINT "Over" SPC(15) "there"
Over            there
Ok
```

Reference

PRINT, LPRINT, SPACE$

SQR

Format

SQR(<x>)

Purpose

To return the square root of <x>. (<x> must be greater than or equal to 0.)

Example

```
10 FOR x=10 TO 25 STEP 5
20 PRINT x, SQR(x)
30 NEXT
RUN
 10        3.162278
 15        3.872984
 20        4.472136
 25        5
Ok
```

STR$

Format

STR$(<x>)

Purpose

To return a string representation the value of <x>.

Example

```
5 REM Arithmetic for kids
10 INPUT "Type a number ";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

Reference

VAL

STRING$

Format

```
STRING$(<I>,<J>)
STRING$(<I>,<X$>)
```

Purpose

To return a string of length <I> whose characters all include ASCII code <J> or the first character of <X$>.

Example

```
10 X$ = STRING$(10,45)
20 PRINT X$ "Monthly Report" X$
RUN
---------- Monthly Report ----------
Ok
```

TAB

Format

```
TAB(<I>)
```

Purpose

To space to position <I> on the terminal. If the current print position is already beyond <I>, TAB goes to that position on the next line. Position 1 is the leftmost screen position. The rightmost position is the width minus 1. (<I> must be in the range 1 to 255.)

TAB can only be used in PRINT and LPRINT statements.

Example

```
10 PRINT "Name" TAB(25) "Amount":PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. Jones", "$25.00"
RUN
Name                    Amount
G.T. Jones              $25.00
Ok
```

Reference

LPRINT, PRINT

TAN

Format

```
TAN(<x>)
```

Purpose

To return the tangent of <x> in radians. TAN(<x>) is calculated in single precision.

If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and program execution continues.

Example

```
10  y=Q*TAN(x)/2
```

TIME$

Format

TIME$ = <string expr>	(Sets the current time.)
<string exp> = TIME$	(Returns the current time.)

Purpose

To set or return the current time.

Remarks

<string expr> is a valid string literal or variable.

TIME$ is a system variable that has a built-in clock.

TIME$ = <string exp> may be in one of the following forms:

hh	Sets the hour. Minutes and seconds default to 00.
hh:mm	Sets the hour and minutes. Seconds default to 00.
hh:mm:ss	Sets the hour, minutes and seconds.

If any of the values are out of the allowed range, an "Illegal function call" error occurs.

<string expr> = TIME$ returns an 8-character string in the form "hh:mm:ss", where hh is the hour (00 to 23), mm is minutes (00 to 59), and ss is seconds (00 to 59).

If <string exp> is not a valid string, a "Type mismatch" error results.

Example

```
100 INPUT "Time(hh.mm.ss) = ";T$
110 TIME$ = T$
120 CLS
130 LOCATE 10,20
140 PRINT "Present time is ";TIME$
150 GOTO 130
```

TIMER

Format

```
<x>=TIMER
```

Purpose

To return the number of seconds that have elapsed since midnight or since the system was reset.

Remarks

Seconds are represented by a single precision number with fractional seconds calculated to the nearest hundredth. The TIMER function can only be read.

Example

```
10 CLS
20 TICS=TIMER
30 x=INT(TICS)
40 LOCATE 10,20:PRINT x
50 GOTO 20
```

USR

Format

```
USR[<digit>](<x>)
```

Purpose

To call the user's assembly language subroutine with the argument <x>.

<digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine.

If <digit> is omitted, USR 0 is assumed.

Example

```
40 B=T*SIN(Y)
50 C=USR(B/2)
60 D=USR(B/3)
.
.
.
```

Reference

DEF USR

VAL

Format

```
VAL(<X$>)
```

Purpose

To return the numeric value of string <X$>. VAL strips leading blanks, tabs, and line feeds from the argument string. For example:

```
VAL("-3")
```

returns -3.

Example

```
10 READ Nam$, City$, State$, Zip$
20 IF VAL(Zip$)<90000 OR VAL(Zip$)>96699 THEN PRINT
   Nam$ TAB(25) "Out of State"
30 IF VAL(Zip$)>=90801 AND VAL(Zip$)<=90815 THEN PRINT
   Nam$ TAB(25) "Long Beach"
.
.
.
```

Reference

STR$

VARPTR

Format

VARPTR(<variable name>)

Purpose

To return the address of the first byte of data identified with <variable name>.

Remarks

A value must be assigned to <variable name> before executing VARPTR. Otherwise, an "Illegal function call" error results. Any type variable name may be used (numeric, string, array). The address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. Use VARPTR(A(0)) when passing an array, so that the lowest addressed element of the array is returned.

NOTE:

Assign simple variables before calling VARPTR for an array because the addresses of the arrays change whenever a new simple variable is assigned.

Example

```
100 x =USR(VARPTR(y))
```

VARPTR$

Format

VARPTR$(<variable name>)

Purpose

To return the address of a variable in character format.

Remarks

The address at which variable data is stored is returned in 3-byte character format. Before executing VARPTR$, assign a value to the variable. The address for the variable is in the following format:

Byte 0	Type.
Byte 1	Low-order byte of variable address.
Byte 2	High-order byte of variable address.

The type, which shows the type of <variable name>, is one of the following:

2	Integer type.
3	Character type.
4	Single-precision real type.
8	Double-precision real type.

The type value which VARPTR$ returns is the same as:

CHR$(<type>) + MKI$(VARPTR(<variable name>))

VARPTR$ is usually used to indicate the variable name in PLAY and DRAW statements. For example, the following may be specified in a PLAY statement:

```
PLAY "xA$;"
```

This example can also be written using the VARPTR$ function:

```
PLAY "x" = VARPTR$(A$)
```

Reference

DRAW, PLAY

Appendixes

Figures

Appendix A. BASIC Disk I/O

This appendix includes disk I/O procedures for the beginning BASIC user. If you have been getting disk-related errors, read through these procedures and examples to make sure you're using disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to MS-DOS requirements.

MS-DOS adds the default extension (.BAS) to the filename specified in a SAVE, RUN, MERGE, or LOAD command.

Program File Commands

A review of the commands and statements used in program file manipulation follows.

```
SAVE "<filename>" [,A]
```

writes to disk the program currently residing in memory. Option A writes the program as a series of ASCII characters. Otherwise, BASIC uses a compressed binary format.

```
LOAD "<filename>" [,R]
```

loads the program from disk into memory. Option R runs the program immediately.

LOAD deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections.

LOAD "<filename>",R is the same as RUN "<filename>",R.

RUN "<filename>" [,R]

loads the program from disk into memory and runs it.

RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.

RUN "<filename>",R is the same as LOAD "<filename>",R.

MERGE "<filename>"

loads the program from disk into memory, but does not delete the current contents of memory. Program line numbers on disk are merged with the line numbers in memory.

If two lines have the same number, only the line from the disk program is saved. The merged program resides in memory and BASIC returns to command level.

KILL "<filename>"

deletes the file from disk.

<filename> may be a program file, or it may be a sequential or random access data file.

NAME "<old filename>" AS "<new filename>"

Changes the name of a disk file.

NAME may be used with program files, random files, or sequential files.

Protected Files

To save a program in encoded binary format, use the SAVE command. For example:

```
SAVE "Myprog",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing.

Disk Data Files

Two types of disk data files may be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random files, but are limited in flexibility and speed when you need to access the data. Data written to a sequential file is a series of ASCII characters stored sequentially, in the order sent. Data is read in the same way.

Statements and functions used with sequential files are:

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

Appendix A

Creating a Sequential File

To create a sequential file and access data in the file:

1. OPEN the file in "O" mode. OPEN "O",#1,"Data"

2. Write data to the file using the PRINT# or WRITE# statement. PRINT#1,A$;B$;C$

Example

```
10 OPEN "O",#1,"Data"
20 INPUT "Name";N$
25 IF N$="Done" THEN END
30 INPUT "Department";D$
40 INPUT "Date hired";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN

Name? Mickey Mouse
Department? Entertainment
Date hired? 01/12/72

Name? Sherlock Holmes
Department? Research
Date hired? 12/03/65

Name? Ebenezer Scrooge
Department? Accounting
Date hired? 04/27/78

Name? Superman
Department? Features
Date hired? 08/16/78

Name? Done
```

The preceding example creates a sequential file ("Data") from information entered at the terminal.

Accessing a Sequential File

To access data in a sequential file, close the file and reopen it in "I" mode.

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

Example

```
10 OPEN "I", #1, "Data"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2) = "78" THEN PRINT N$
40 GOTO 20
RUN

Ebeneezer Scrooge
Superman
Input past end in 20
Ok
```

This example accesses the file that was created and displays the names of everyone hired in 1978.

To read data from a sequential file into the program, use the INPUT# statement.

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

The previous example reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid this error, insert line 15, which uses the EOF function to test for the end of file:

```
15 IF EOF(1) THEN END
```

and change line 40 to

```
GOTO 15
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example,

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

writes numeric data to disk without explicit delimiters. The comma at the end of the format string separates the items in the disk file.

Use the LOC function with a sequential file to return the number of sectors that have been written to or read from the file since it was opened.

If you have a sequential file on disk and want to add more data, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. Instead, open the file for APPEND.

Random Files

Creating and accessing random files requires more program steps than creating and accessing sequential files, but there are advantages to using random files. Random files require less room on the disk because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

Data in random files can be accessed randomly, anywhere on the disk, so it is not necessary to read through all the information. Data is stored and accessed in distinct units called records and each record is numbered.

Statements and functions used with random files are:

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

Creating a Random File

To create a random file:

1. Open the file for random access ("R") mode. If the record length is omitted, the default is 128 bytes.

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

2. Use the FIELD statement to allocate space in the random buffer for variables to be written to the random file.

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

3. Use LSET to move the data into the random buffer.

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

   Numeric values must be made into strings when placed in the buffer. To do this, use MKI$ to convert an integer value to a string, MKS$ for a single precision value, and MKD$ for a double precision value.

4. Use the PUT statement to write the data from the buffer to the disk.

```
PUT$1, CODE%.
```

Information entered at the terminal is written to a random file in the following example. Each time the PUT statement is executed, a record is written to the file. The 2-digit code in line 30 becomes the record number.

Example

```
10 OPEN "R",#1,"File", 32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-Digit code";Code%.
40 INPUT "Name";X$
50 INPUT "Amount";AMT
60 INPUT "Phone";Tel$:PRINT
70 LSET N$ = X$
80 LSET A$ = MKS$(AMT)
90 LSET P$ = Tel$
100 PUT #1,Code%
110 GOTO 30
```

NOTE:

Do not use a fielded string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Accessing a Random File

To access a random file:

1. OPEN the file in "R" mode.

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

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

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

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,Code%
```

   The data in the buffer may now be accessed by the program.

```
PRINT N$
```

4. Convert numeric values to numbers using CVI for integers, CVS for single precision values, and CVD for double precision values.

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

NOTE:

If a program performs both input and output on the same random file, often you can use just one OPEN statement and one FIELD statement.

Example 1 accesses the random file (File). When the 3-digit code is entered at the terminal, the information associated with that code is read from the file and displayed on the screen.

Example 1

```
10 OPEN "R", #1, "File",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-Digit code";code%
40 GET #1, code%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

The LOC function, with random files, returns the current record number. The current record number is one plus the last record number that was used in a GET or PUT statement. For example,

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in the file is higher than 50.

Example 2 is an inventory program that illustrates random file access. In this program, the record number is used as the part number. It is assumed that the inventory contains no more than 100 different part numbers.

Lines 900 through 960 initialize the data file by using CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130 through 220 display the different inventory functions the program performs. When you enter the desired function number, line 230 branches to the appropriate subroutine.

Example 2

```
120 OPEN"R",#1,"Inven.dat",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "Functions:":PRINT
135 PRINT 1,"Initialize File"
140 PRINT 2,"Create a new entry"
150 PRINT 3,"Display inventory for one part"
160 PRINT 4,"Add to stock"
170 PRINT 5,"Subtract from stock"
180 PRINT 6,"Display all items below reorder level"
220 PRINT:PRINT:INPUT"Function";Function
225 IF (Function<1)OR(Function>6) THEN PRINT "Bad func-
    tion number":GOTO 130
230 ON Function GOSUB 900,250,390,480,560,680
240 GOTO 130
250 REM Build new entry
260 GOSUB 840
270 IF ASC(F$)< >255 THEN INPUT"Overwrite";A$:IF A$< >"Y"
    THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "Description";DESC$
300 LSET D$=Desc$
310 INPUT "Quantity in stock";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "Reorder level";R%
340 LSET R$=MKI$(R%)
350 INPUT "Unit price";P
```

```
360 LSET P$ = MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM Display entry
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "Null entry":RETURN
420 PRINT USING "Part number ###";Part%
430 PRINT D$
440 PRINT USING "Quantity on hand ####";CVI(Q$)
450 PRINT USING "Reorder level #####";CVI(R$)
460 PRINT USING "Unit price $$##.##";CVS(P$)
470 RETURN
480 REM Add to stock
490 GOSUB 840
500 IF ASC(F$) = 255 THEN PRINT "Null entry":RETURN
510 PRINT D$:INPUT "Quantity to add";A%
520 Q% = CVI(Q$) + A%
530 LSET Q$ = MKI$(Q%)
540 PUT#1,Part%
550 RETURN
560 REM Remove from stock
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "Null entry":RETURN
590 PRINT D$
600 INPUT "Quantity to subtract";S%
610 Q% = CVI(Q$)
620 IF (Q% - S%)<O THEN PRINT "Only";Q%;" in stock":GOTO 600
630 Q% = Q% - S%
640 IF Q% =<CVI(R$) THEN PRINT "Quantity now";Q%;"Reorder
    level";CVI(R$)
```

```
650 LSET Q$ = MKI$(Q%)
660 PUT#1,Part%
670 RETURN
680 REM Display items below reorder level
690 FOR I = 1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;"Quantity";CVI(Q$)TAB(50)"Reorder
    level";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "Part number";Part%
850 IF(Part%<1)OR(Part%>100) THEN PRINT "Bad part number":GOTO
    840 ELSE GET#1,Part%:RETURN
890 END
900 REM Initialize file
910 INPUT "Are you sure ";B$:IF B$< >"Y" THEN RETURN
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Appendix B. Communications

This appendix describes the BASIC statements needed to implement communications through the RS-232-C interface.

Communication File

The communication file is opened by using OPEN COM. The input/output buffer is allocated to this file. Use the CLOSE statement to close the file.

Communication I/O Unit

A communication unit is opened as a file. All input/output statements valid for the diskette files are valid for communications. The following sequential input statements are the same as those for disk files:

INPUT# file number
LINE INPUT# file number
INPUT$

The following sequential output statements are also the same as those for disk files:

PRINT# file number
PRINT# file number USING
WRITE# file number

GET and PUT Statements

GET and PUT communication statements differ somewhat from those used for flexible disk files. They are used to input and output fixed-length records to or from the communication file.

The number of bytes to be transmitted or received is written to the specified location of the record number to be accessed. The value should not exceed the range of record numbers specified.

To accommodate large files with short record lengths, GET and PUT statements allow record numbers in the range 1 to 16,777,215.

I/O Functions

Difficulties with asynchronous communications may occur because processing is performed as each character is entered. It is often necessary to prevent the data buffer from overflowing by sending XOFF(CHR$(19)) and XON(CHR$(17)) statements from another computer. XOFF stops the transmission to the other computer. XON signals "Restart the transmission." Other codes may be used if they do not overlap with data.

BASIC provides three functions to detect the input condition:

LOC(<x>)	Number of characters in the input buffer to be read. If this number is greater than 255, 255 is returned.
LOF(<x>)	Number of characters in the free area of the input buffer is returned. This value is equal to n–LOC(f). (N is the size of the communication buffer set by the /C: option.) The default value of n is 256.
EOF(<x>)	If the input buffer is empty, true (–1) is returned. If there are any characters to be read, false (0) is returned.

NOTE:

If a read function is attempted when the input buffer is full, or 0 is returned by LOF (file number), a communication buffer overflow occurs.

INPUT$ Function

With INPUT$, all characters are assigned to strings. INPUT$ (x,#y) means that x characters are returned from file y.

When reading the communication file, INPUT$ is more convenient to use than INPUT# or LINE INPUT#. This is because all ASCII characters are valid in communications.

INPUT# terminates the operation when a comma or carriage return is encountered. LINE INPUT# terminates the operation when a carriage return is encountered.

The following example shows the most effective reading from the communication file:

```
10 WHILE NOT EOF (1)
20 B$ = INPUT$(LOC(1),#1)
.
.    (Data in B$ is processed.)
.
100 WEND
```

If the input buffer contains data, the number of characters in the buffer is returned and assigned to B$.

Appendix C. BASIC Assembly Language Subroutines

The CALL statement and the USR function provide interfacing with assembly language subroutines.

The CALL statement is recommended for interfacing iAPX86 machine language programs, or iAPX286 Real Address Mode machine language programs, with BASIC. It is compatible with more languages than is the USR function call, produces more readable source codes, and passes multiple arguments.

Memory Allocation

In addition to the interpreter code area, BASIC uses up to 64K bytes of memory beginning at its data segment (DS).

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack can be set up for the assembly language subroutine. You must restore BASIC's stack, however, before returning from the subroutine.

Load the assembly language subroutine into memory by using the operating system or the POKE statement. Routines may be assembled and linked, but not loaded.

To load the program file, skip the first 512 bytes of the output file, then read the rest of the file. (The subroutines must not contain any long references.)

NOTE:

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, use the /M: option to enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s).

The CALL Statement

The CALL statement is recommended for interfacing machine language programs with BASIC.

Format

CALL <variable name> [(<argument list>)]

<variable name> contains the segment offset that is the starting point in memory of the subroutine being called.

<argument list> contains the variables or constants, separated by commas, to be passed to the routine.

Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the 2-byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.
2. BASIC's return address code segment (CS), and offset (IP) are pushed onto the stack.

3. Control is transferred to the user's routine with the segment address specified in the last DEF SEG statement and the offset specified in <variable name>.

Figures C-1 and C-2 illustrate the state of the stack at the time of the CALL statement and the condition of the stack during execution of the called subroutine.

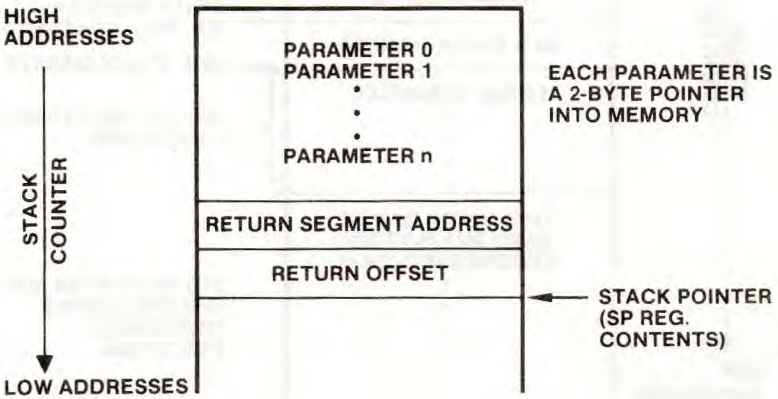


Figure C-1. Stack Layout When CALL Statement is Activated

The user's routine now has control. You can reference parameters by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to (BP).

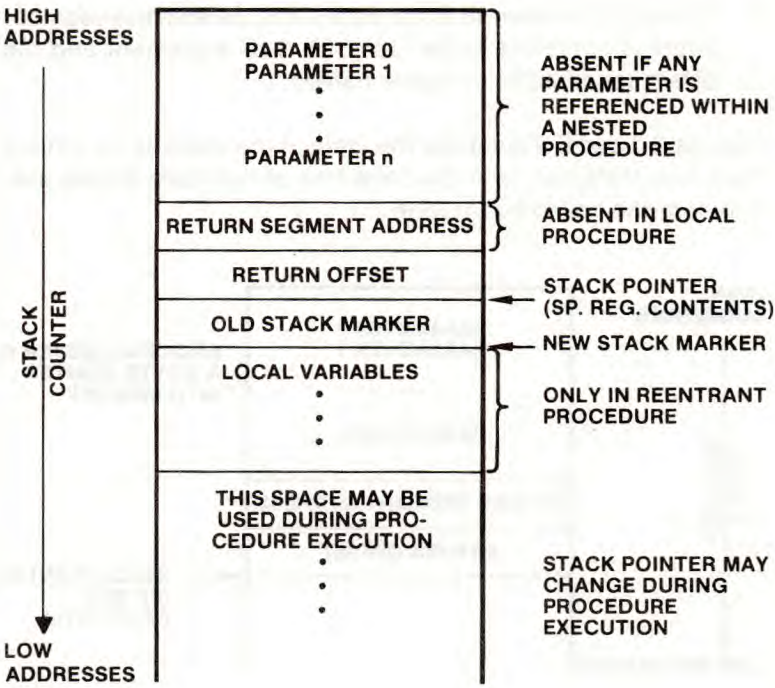


Figure C-2. Stack Layout During Execution of a CALL Statement

When coding a subroutine:

1. The called routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.
2. The called program must know the number and length of the parameters passed. References to parameters are positive offsets added to BP (assuming the called routine moved the current stack pointer into BP).

3. The called routine must do a RET <n> to adjust the stack to the start of the calling sequence. <n> is two times the number of parameters in the argument list.
4. Values are returned to BASIC by including the variable name to receive the result in the argument list.
5. If the argument is a string, the parameter's offset points to three bytes called the string descriptor. Byte 0 contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper eight bits of the string starting address in string space.
6. Strings may be altered by user routines, but the length must not be changed. BASIC cannot correctly manipulate strings if their lengths are changed by external routines.

Example

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

Line 100 sets the segment to 8000 hexadecimal. The value of variable FOO is added into the address as the low word after the DEF SEG value is shifted left eight bits. (This is a function of the microprocessor, not BASIC.) Here, FOO is set to &H7FA, so the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

NOTE:

If the argument is a string literal in a program, the string descriptor points to program text. Be careful not to alter or destroy your program this way.

To avoid unpredictable results, add +" " to the string literal in the program.

Example

```
20 A$="BASIC"+" "
```

This forces the string literal to be copied into string space. Now the string may be changed without affecting the program.

The following assembly language sequence demonstrates access of the parameters passed, storing a return result in the variable C.

```
MOV     BP,SP       ;Get current stack position in BP
MOV     BX,6[BP]    ;Get address of B$ string descriptor
MOV     CL,[BX]     ;Get length of B$ in CL
MOV     DX,1[BX]    ;Get addr of B$ text in DX
        .
        .
        .
MOV     SI,8[BP]    ;Get address of A in SI
MOV     DI,4[BP]    ;Get pointer to C in DI
MOVS    WORD        ;Store variable A in C
RET     6           ;Restore stack
```

NOTE:

The called program must know the variable type for numeric parameters passed. In the preceding example, the instruction,

```
MOVS WORD
```

copies only two bytes. This is fine if variables A and C are integers. Copy four bytes if they are single precision and eight bytes if they are double precision.

USR Function Calls

Although the CALL statement is recommended for calling assembly language subroutines, the USR function call is available for compatibility with previously written programs.

The USR function allows assembly language subroutines to be called in the same way BASIC intrinsic functions are called.

Format

```
USR[<digit>][(argument)]
```

where digit is 0 to 9 and argument is any numeric or string expression.

<digit> specifies which USR routine is being called (refer to the DEF USR statement). If <digit> is omitted, USR0 is assumed.

Arguments are discussed in the following paragraphs.

A DEF SEG statement must be executed prior to a USR function call to ensure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (Refer to DEF SEG.)

For each USR function call, a corresponding DEF USR statement must be executed to define the USR call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is performed, register [AL] contains a value that specifies the type of the argument given. The value in [AL] may be one of the following:

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 number, the [BX] register pair point to the floating point accumulator (FAC) where the argument is stored:

FAC is the exponent minus 128. The binary point is to the left of the most-significant bit of the mantissa.

FAC-1 contains the highest seven bits of the mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive, 1 = negative).

If the argument is an integer:

FAC-2 contains the upper eight bits of the argument.

FAC-3 contains the lower eight bits of the argument.

If the argument is a single precision floating-point number:

FAC-2 contains the middle eight bits of the mantissa.

FAC-3 contains the lowest eight bits of the mantissa.

If the argument is a double precision floating-point number:

FAC-7 through FAC-4 contains four more bytes of the mantissa (FAC-7 contains the lowest eight bits.)

If the argument is a string, the [DX] register pair points to three bytes called the string descriptor. Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper eight bits of the string starting address in BASIC's data segment.

Example

```
110 DEF USR0=&H8000
120 x=5
130 y=USR0(x)
140 PRINT y
```

Assumes user gave /M:32767.
Note that x is single precision.

The type of variable (numeric or string) receiving the function call must be consistent with the argument passed. (It may be set to an integer by calling MAKINT in the user's routine before returning to BASIC.)

NOTE:

If the argument is a string literal in a program, the string descriptor points to program text. Be careful not to alter or destroy your program this way. (Refer to the CALL statement.)

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument.

When FRCINT or MAKINT is called and the subroutine terminates with a return, ES, DS, and SS must have the same value they had when the subroutine was entered. These registers point to BASIC's data segment.

In the following example, an assembly language routine is loaded to multiply the argument passed by 2 and return an integer result.

(Be sure your programs are defined by a PROC FAR statement.)

Example

```
DOUBLE          SEGMENT
                ASSUME  CS:DOUBLE

FRCINTOFFSET    EQU     103H
MAKINTOFFSET    EQU     107H
FRCINT          LABEL   DWORD
                DW      FRCINTOFFSET
FRCSEG          DW      ?

MAKSEG          LABEL   DWORD
                DW      MAKINTOFFSET
MAKSEG          DW      ?

USRPRG          PROC    FAR
                POP     SI
                POP     AX          ;Recover BASIC's CS
                PUSH    AX
                PUSH    SI
                MOV     FRCSEG,AX   ;Set segment for
                                     long indirect CALL

                MOV     MAKSEG,AX
                CALL    FRCINT      ;Force arg in
                                     FAC to int in [BX]
                ADD     BX,BX       ;[BX]=[BX]*2
                CALL    MAKINT      ;Put result back in FAC
                RET                 ;Long return to BASIC
USRPRG          ENDP

DOUBLE          ENDS
```

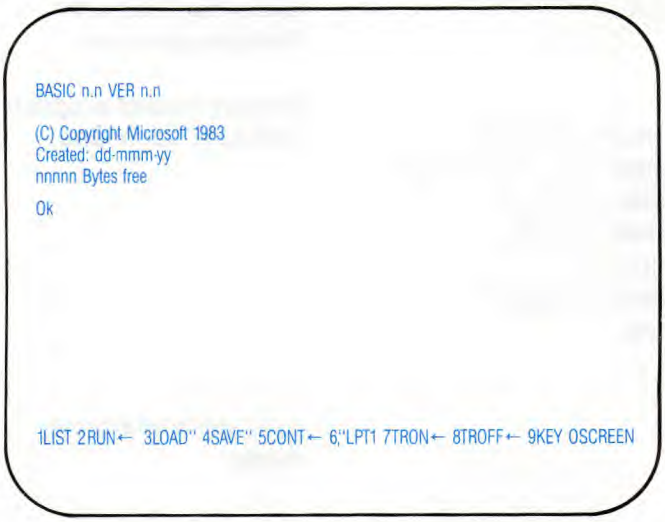
Appendix D. Program Execution Example

When MS-DOS displays the A> prompt, type BASIC and press the **Return** or the **Enter** key:

```
A>BASIC
```

The following display appears on the screen. (Free bytes depend upon memory size.)

In the following example, system responses are indicated in blue.



Example

```
Ok
10 PRINT"***START***"
20 PRINT"*I*","*I+I*","*IxI*"
30 FOR I=1 TO 10
40 PRINT I,I+I,I*I
50 PRINT "***END***"
60 END
```

Command level of BASIC. Make source program.

```
RUN
***START***
*I*  *I+I*  *IxI*
FOR Without NEXT in 30
Ok
45 NEXT I
RENUM
Ok
```

Execute in interpreter mode.

Error message.

Add line number 45 and renumber lines with RENUM command.

```
LIST
10 PRINT"***START***"
20 PRINT "*I*", "*I+I*","*IxI*"
30 FOR I=1 TO 10
40 PRINT I, I+I,I*I
50 NEXT I
60 PRINT "***END***"
70 END
Ok
```

Display revised program with LIST command.

```
RUN
```

Execute in interpreter mode.

```
***START***
    *I*       *I+I*     *IxI*
     1         2         1
     2         4         4
     3         6         9
     4         8         16
     5         10        25
     6         12        36
     7         14        49
     8         16        64
     9         18        81
     10        20        100
***END***
Ok

SAVE "test

Ok
FILES
A:\
TEST     .BAS
xxxxxx Bytes free
Ok
SYSTEM


A>DIR  test.bas


Volume in drive A has no label
Directory of A:\

TEST   .BAS    113   4-25-85   10:57a
       1 FILE(S) xxxxxx bytes free
```

Save program.

Check filename with FILES command.

Return to MS-DOS with SYSTEM command.

Check file with DIR command.

```
A>CHKDSK
```

Check status of use of diskette with CHKDSK command.

```
xxxxxx bytes total disk space
 xxxxx bytes in y hidden files
xxxxxx bytes in z user files
  xxxx bytes available on disk

xxxxxx bytes total memory
xxxxxx bytes free
```

```
A>BASIC
BASIC n.n VER. nn

(C) Copyright Microsoft 1983
Created: xx-xxx-xx
xxxxx Bytes free
Ok
```

Start BASIC.

```
FILES"test .bas"
A:\

TEST .BAS
xxxxxx Bytes free
Ok
```

Check filename with FILE command.

```
LOAD"test"
Ok
```

Load test.bas file.

```
LIST
```

Display program with LIST command.

```
10 PRINT"***START***"
20 PRINT"*I*","*I+I*","*IxI*"
30 FOR I=1 TO 10
40 PRINT I,I+I,I*I
50 NEXT I
60 PRINT"***END***"
70 END
Ok
```

```
EDIT 30
```

Revise line number 30 with EDIT command.

```
30 FOR I=1 TO 10

30 FOR I=1 TO 20
```

```
LIST.
```

Display revised program with LIST command.

```
10 PRINT"***START***"
20 PRINT"*I*","*I+I*","IxI*"
30 FOR I=1 TO 20
40 PRINT I,I+I,I*I
50 NEXT I
60 PRINT"***END***"
70 END
Ok
```

```
RUN                          Execute program in intepreter mode.
***START***
*I*         *I+I*     *IxI*
 1          2         1
 2          4         4
 3          6         9
 4          8         16
 5          10        25
 6          12        36
 7          14        49
 8          16        64
 9          18        81
 10         20        100
 11         22        121
 12         24        144
 13         26        169
 14         28        196
 15         30        225
 16         32        256
 17         34        289
 18         36        324
 19         38        361
 20         40        400
***END***
Ok

SAVE"test                    Save "test.bas" program.
Ok

FILES"test.bas"              Check filename with
                             FILES command.
A:\
TEST    .BAS
xxxxxx Bytes free
Ok

RESET                        Rewrite directory
Ok                           information in diskette
SYSTEM                       with RESET command.
```

Appendix E. Converting Programs to SPERRY BASIC

Programs written in another version of BASIC may require some minor adjustments before they can be run with SPERRY BASIC. Information for converting other programs to SPERRY BASIC is included in this appendix.

String Dimensions

Delete all statements used to declare the length of strings. Convert a statement such as DIM A$ (I,J), which sizes a string array for J elements of length I, to DIM A$ (J).

Some BASIC versions use a comma or ampersand (&) for string concatenation. These characters must be changed to a plus sign, the operator for SPERRY BASIC string concatenation.

In this version of SPERRY BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Convert forms such as A$ (I), or A$ (I,J) as follows:

SPERRY BASIC	Other Versions
X$ = MID$(A$,I,1)	X$ = A$(I)
X$ = MID$(A$,I,J-I + 1)	X$ = A$(I,J)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

SPERRY BASIC	Other Versions
MID$(A$,I,1) = X$	A$(I) = X$
MID$(A$,I,J – I + 1) = X$	A$(I,J) = X$

Multiple Assignments

Some versions of BASIC allow statements of the following form to set B and C equal to zero.

```
10 LET B=C=0
```

SPERRY BASIC interprets the second equal sign as a logical operator and sets B equal to –1 if C equals 0. Convert this statement to two assignment statements:

```
10 B=0:C=0
```

Multiple Statements

Some versions of BASIC use a backslash (\) to separate multiple statements on a line.

With SPERRY BASIC, use a colon (:) to separate statements on a line.

MAT Functions

Programs that use MAT functions must be converted by using FOR. . .NEXT loops.

Appendix F. Error Codes and Error Messages

Error Message	Error Code	Meaning
Bad file mode	54	An attempt was made to use PUT, GET or LOF with a sequential file, to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.
Bad file name	64	An illegal form was used for the filename in LOAD, SAVE, KILL, or OPEN (usually a filename with too many characters).
Bad file number	52	A statement or command referenced a file with a file number that is not open or is out of the range of file numbers specified at initialization.
Bad record number	63	In a PUT or GET statement, the specified record number is greater than the maximum allowed (32767) or equals zero.
Can't continue	17	An attempt was made to continue a program that has halted due to an error, has been modified during a break in execution, or does not exist.
Can't continue after SHELL	–	Upon returning from a child process, the SHELL statement discovers that there is not enough memory for BASIC to continue. BASIC closes open files and exits to MS-DOS.

Appendix F

Error Message	Error Code	Meaning
Communication buffer overflow	69	The data transmitted from the communication port overflowed the input buffer.
Device fault	25	The hardware (printer, disk, etc.) is out of order.
Device I/O error	57	An I/O error has occurred during disk I/O operation. The operating system cannot recover from the error.
Device timeout	24	The BASIC program cannot get a data or ready signal from a device.
Device unavailable	68	An attempt was made to open the file for an unavailable device or the I/O device specified is unavailable.
Direct statement in file	66	A direct statement is encountered while loading an ASCII-format file. The LOAD is terminated.
Disk full	61	All disk storage space is in use.
Disk medium error	72	An abnormal condition occurred on the disk drive or diskette. (Usually, this is caused by scratches on the diskette.)
Disk not ready	71	The door of the diskette drive is open, or the drive is not loaded with a diskette.
Disk write protected	70	An attempt was made to write data to a write-protected diskette.

Error Message	Error Code	Meaning
Division by zero	11	A division by zero is encountered in an expression or the result of an operation of involution is zero raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of division (or positive machine infinity is supplied as the result of the involution) and execution continues.
Duplicate definition	10	Two DIM statements were given for the same array, or a DIM statement was given for an array after the default dimension of 10 had been established for that array.
Field overflow	50	A FIELD statement has attempted to allocate more bytes than were specifed for the record length or a random file.
File already exists	58	The filename specified in a NAME statement is identical to a file-name already in use on the disk.
File already open	55	A sequential output mode OPEN statement was issued for a file already open, or a KILL statement was given for an open file.
File not found	53	A LOAD, KILL, or OPEN statement referenced a file that does not exist on the current disk.
FOR without NEXT	26	A FOR is encountered without a matching NEXT.

Error Message	Error Code	Meaning
Illegal direct	12	A command that is illegal in direct mode was entered as a direct mode command.
Illegal function call	5	An out-of-range parameter was passed to a math or string function. This error may also occur as the result of: 1. A negative or unreasonably large subscript. 2. A negative or zero argument with LOG. 3. A negative argument to SQR. 4. A negative mantissa with a non-integer exponent. 5. A USR function call for which the starting address has not yet been given. 6. An improper argument to INP, INSTR, LEFT$, MID$, ON...GOTO, OUT, PEEK, POKE, SPACE$, SPC, STRING$, TAB, or WAIT.
Internal error	51	An internal malfunction occurred in disk BASIC. Report the conditions under which the message appeared to your Sperry representative.
Input past end	62	An INPUT statement was executed after all the data in the file had been input or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.
Line buffer overflow	23	An attempt was made to execute a line with too many characters.
Missing operand	22	An expression contains an operator with no operand.

Error Message	Error Code	Meaning
NEXT without FOR	1	A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
No RESUME	19	An error trapping routine was entered without a RESUME statement.
Out of data	4	A READ statement is executed when no DATA statements with unread data remaining in the program exist.
Out of memory	7	A program is too large, has too many FOR or GOSUB loops, contains too many variables, or includes expressions that are too complicated.
Out of paper	27	Printer is out of paper, not connected, or not receiving power.
Out of string space	14	String variables have caused BASIC to exceed the amount of remaining free memory. (BASIC allocates string space until it runs out of memory.)
Overflow	6	The result of a calculation is too large to be represented in BASIC's number format.
Path/file access error	75	An attempt was made to use a path or filename with an inaccessible file. For example, you may have tried to open a read-only file for writing or tried to remove the current directory.

Appendix F

Error Message	Error Code	Meaning
Path not found	76	MS-DOS is unable to find a path specified in an OPEN, MKDIR, CHDIR, or RMDIR statement.
Rename across disks	74	The incorrect disk was specified in an attempt to rename a file.
RESUME without error	20	A RESUME statement is encountered before an error trapping routine has been entered.
RETURN without GOSUB	3	A RETURN statement for which there is not a previous, unmatched GOSUB statement is encountered.
String formula too complex	16	A string expression is too long or too complex. Divide the expression into smaller expressions.
String too long	15	An attempt was made to create a string more than 255 characters long.
Subscript out of range	9	An array element is referenced, with a subscript outside the dimensions of the array or with the wrong number of subscripts.
Syntax error	2	A line that contains an incorrect sequence of characters is encountered (unmatched parentheses, a misspelled command or statement, incorrect punctuation, etc.).
Too many files	67	An attempt was made to create a new file (using SAVE or OPEN) when all 255 directory entries were full.

Error Message	Error Code	Meaning
Type mismatch	13	A string variable name was assigned a numeric value (or vice versa) or a function that expects a numeric argument was given a string argument (or vice versa).
Undefined line	8	A GOTO, GOSUB, IF...THEN... ELSE, or DELETE statement contains a reference to a nonexistent line.
Undefined user function	18	A USR function was called before the function definition (DEF statement) was given.
Unprintable error	21	An error message is not available for the existing error condition.
You cannot SHELL to BASIC	–	During initialization, BASIC discovers that it is being run as a child task. BASIC terminates and returns control to the parent copy of BASIC.
WEND without WHILE	30	A WEND is encountered without a matching WHILE statement.
WHILE without WEND	29	A WHILE statement does not have a matching WEND statement.

Appendix G. Mathematical Functions

Functions not intrinsic to BASIC and their equivalents in BASIC follow:

Function	BASIC Equivalent
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(-X*X + 1))
INVERSE COSINE	ARCCOS(X) = -ATN (X/SQR(– X*X + 1)) + 1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X – 1)) + SGN(SGN(X) – 1)*3.141593
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X – 1)) + SGN(SGN(X) – 1)*3.141593
INVERSE COTANGENT	ARCCOT(X) = ATN(-X) + 1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X) + EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = (EXP(X) – EXP(-X))/(EXP(X) + EXP(– X))
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X) + EXP(– X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X) – EXP(– X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(– X)/ EXP(X) – EXP(– X))*2 + 1

Appendix G

Function	BASIC Equivalent
INVERSE HYPERBOLIC SINE	ARCSINH(X) = LOG(X + SQR(X*X + 1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X) = LOG(X + SQR(X*X – 1))
INVERSE HYPERBOLIC TANGENT	ARCTANH(X) = LOG((1 + X)/(1 – X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X) = LOG((SQR(– X*X + 1) + 1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) = LOG((SGN(X)*SQR(X*X + 1) + 1)/X)
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X) = LOG((X + 1)/(X – 1))/2

Appendix H. ASCII Character Codes

The following codes have special meanings in BASIC:

07	Beep
09	Tab
10	Line feed
11	Home
12	Form feed
13	Carriage return
28	Cursor right
29	Cursor left
30	Cursor up
31	Cursor down

Tables H-1 and H-2 show ASCII codes in decimal and corresponding characters. These characters are displayed using PRINT CHR$(<n>), where n is an ASCII code.

Appendix H

Table H–1. Complete ASCII Character Codes (Part 1 of 4)

ASCII VAlue	Character	Control Character	ASCII Value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	▪	BS	040	(
009	(tab)	HT	041	)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016		DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	‼	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

Table H-1. Complete ASCII Character Codes (Part 2 of 4)

ASCII Value	Character	ASCII Value	Character
064	@	096	`
065	A	097	a
066	B	098	b
067	C	099	c
068	D	100	d
069	E	101	e
070	F	102	f
071	G	103	g
072	H	104	h
073	I	105	i
074	J	106	j
075	K	107	k
076	L	108	l
077	M	109	m
078	N	110	n
079	O	111	o
080	P	112	p
081	Q	113	q
082	R	114	r
083	S	115	s
084	T	116	t
085	U	117	u
086	V	118	v
087	W	119	w
088	X	120	x
089	Y	121	y
090	Z	122	z
091	[	123	{
092	\	124	|
093	]	125	}
094	^	126	~
095	_	127	Δ

Table H-1. Complete ASCII Character Codes (Part 3 of 4)

ASCII Value	Character	ASCII Value	Character
128	Ç	160	á
129	ü	161	í
130	é	162	ó
131	â	163	ú
132	ä	164	ñ
133	à	165	Ñ
134	å	166	ª
135	ç	167	º
136	ê	168	¿
137	ë	169	⌐
138	è	170	¬
139	ï	171	½
140	î	172	¼
141	ì	173	¡
142	Ä	174	«
143	Å	175	»
144	É	176	▒
145	æ	177	▒
146	Æ	178	▓
147	ô	179	│
148	ö	180	┤
149	ò	181	╡
150	û	182	╢
151	ù	183	╖
152	ÿ	184	╕
153	Ö	185	╣
154	Ü	186	║
155	¢	187	╗
156	£	188	╝
157	¥	189	╜
158	₧	190	╛
159	ƒ	191	┐

Table H-1. Complete ASCII Character Codes (Part 4 of 4)

ASCII Value	Character	ASCII Value	Character
192		224	
193		225	
194		226	
195		227	
196		228	
197		229	
198		230	
199		231	
200		232	
201		233	
202		234	
203		235	
204		236	
205		237	
206		238	
207		239	
208		240	
209		241	
210		242	
211		243	
212		244	
213		245	
214		246	
215		247	
216		248	
217		249	
218		250	
219		251	
220		252	
221		253	
222		254	
223		255	(blank 'FF')

Appendix H

Table H-2. SPERRY Personal Computer ASCII Character Codes

x / y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		▶		0	@	P	‘	p	Ç	É	á	▒	└	╨	α	≡
1	☺	◀	!	1	A	Q	a	q	ü	æ	í	▓	┴	╤	ß	±
2	☻	↕	"	2	B	R	b	r	ë	Æ	ó	▒	┬	╥	Γ	≥
3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
7		↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	Θ	·
A		→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
B		←	+	;	K	[	k	{	ï	¢	½	╗	╦	■	δ	√
C		∟	,	<	L	\	l	¦	î	£	¼	╝	╠	▪	∞	n
D		↔	−	=	M	]	m	}	ì	¥	¡	╜	═	▌	ø	²
E	ß	▲	.	>	N	^	n	~	Ä	₧	«	╛	╬	▐	∈	■
F	☼	▼	/	?	O	_	o	⌂	Å	ƒ	»	┐	╧	▀	∩	

Extended Codes

For a combined key input or a special key input, the INKEY$ variable returns an extended code consisting of two bytes. The first byte returned is for &H00 and the second byte is for one of the codes shown in Table H-3. The 2-byte code is received by INKEY$. If the first byte is found to contain &H00, determine the entered key by the code in the second byte.

Some key codes may be slightly different, depending on which keyboard you are using.

The following example illustrates the expanded 2-byte code returned for the special key inputs shown in Table H-3.

Example

```
100 PRINT "Press a key"
110 A$=INKEY$:IF A$=""THEN 110
120 PRINT LEN (A$)
130 PRINT ASC (RIGHT$ (A$,1))
140 GOTO 100
```

Table H-3. Extended Codes

Second Code	Meaning
3	(Null character) NUL
15	(Shift tab) "|<–"
16–25	Alt Q,W,E,R,T,Y,U,I,O,P
30–38	Alt A,S,D,F,G,H,J,K,L
44–50	Alt Z,X,C,V,B,N,M
59–68	Function keys F1 through F10 (when disabled as soft keys)
71	Home
72	(Cursor up) ↑
73	Pg Up
75	(Cursor left) <–
77	(Cursor right) –>
79	End
80	(Cursor down) ↓
81	Pg Dn
82	Ins
83	Del
84–93	Function 11–20 (Shift F1 through F10)
94–103	Function 21–30 (Ctrl F1 through F10)
104–113	Function 31–40 (Alt F1 through F10)
114	Ctrl Prt Sc
115	(Previous word) Ctrl <–
116	(Next word) Ctrl –>
117	Ctrl End
118	Ctrl Pg Dn
119	Ctrl Home
120–131	Alt 1,2,3,4,5,6,7,8,9, –, =
132	Ctrl Pg Up

Appendix I. Syntax List

BASIC commands, statements, functions, and system variables are summarized in this appendix for your reference.

Commands

Statements

Functions and Variables

Appendix I

Commands

The slash (/) is used to indicate alternate usage. For example, the notation KEY ON/OFF/LIST means that ON, OFF, or LIST may be used: KEY ON, KEY OFF, or KEY LIST.

File Commands

BLOAD filename, offset	Loads a file anywhere in user memory.
BSAVE filespec,offset length	Saves portions of the user's memory on the specified device.
CHDIR "directory\ subdirectory\filename"	Changes the current directory.
FILES "filespec"	Displays filenames on the disk.
KILL "filespec"	Deletes a file from disk.
LOAD "filename",R	Loads a program file from a disk into memory.
MERGE "filename"	Merges a program file into the program in memory.
MKDIR "directory\ subdirectory\filename"	Creates a directory on a specified diskette.
NAME "old filename" AS "new filename"	Changes the name of a disk file.
RESET	Closes all disk files and deletes all system buffers.

RMDIR "directory\subdirectory\filename"	Removes a directory from a specified diskette.
SAVE "filename", A SAVE "filename", P	Saves a program file on disk.

Output to Printer Commands

LCOPY	Prints screen text.
LLIST line number-line number	Lists all or part of the program currently in memory.

Program Creation Commands

AUTO line number, increment	Generates a line number automatically.
CLEAR expression 1, expression 2	Sets all variables and stacks to initial value.
DELETE line number-line number	Deletes program lines.
EDIT line number	Enters edit mode at the specified line.
LIST line number-line number, device	Lists a program on the screen.
NEW	Deletes the program currently in memory and clears all variables.
RENUM new number, old number, increment	Renumbers program lines.

Program Execution Commands

CONT	Continues program execution.
RUN "filename", R	Loads a file from disk into memory and runs it.
RUN line number	Runs the program in current memory.

Return to MS-DOS Command

SYSTEM	Exits BASIC and returns to MS-DOS.

Statements

Child Process Handling Statements

ENVIRON parameter	Modifies parameters in the BASIC environment string table.
SHELL parameter	Executes a child process.

Communication Statements

GET # file number, n bytes	Gets the permissible fixed-length I/O for communication files.
COM(n) ON/OFF/STOP	Enables and disables the trapping of communications.
ON COM(n) GOSUB line	Defines the starting line of the number subroutine used when data arrives at the communication buffer.

OPEN "COM n: options" AS # file number LEN = number	Opens a communication file.
PUT file number, n bytes	Allows fixed-length I/O for communication files.

Debugging Statement

TRON, TROFF	Enables (or disables) a trace of the execution of program statements.

Device Driver Handling Statements

IOCTL #n,string	Sends a control data string to a character device driver.

Error Handling Statements

ERROR integer expression	Simulates the occurrence of a BASIC error or allows error codes to be defined by the user.
ON ERROR GOTO line number	Enables error trapping and specifies the first line of the error handling subroutine.
RESUME line number/NEXT/0	Continues program execution after an error recovery procedure.

Function Key Statements

KEY key number, string expression	Defines and displays the function key assignment text.

KEY ON/OFF/LIST	Displays function keys and turns display off.
KEY (key number) ON/OFF/STOP	Enables, disables, or terminates interrupts caused by a specified key.
ON KEY key number GOSUB line number	Defines the starting line of the subroutine used when a KEY interrupt occurs.

Input/Output (File) Statements

CLOSE # file number, # file number	Concludes I/O to a disk file.
FIELD # file number, field width AS string variable	Allocates space for variables in a random file buffer.
GET # file number, record number	Reads a record from a random disk file into a random buffer.
INPUT # file number, variable list	Reads data from a sequential disk file and assigns it to program variables.
LINE INPUT # file number, string variable	Reads an entire line from a sequential disk data file to a string variable.
LSET string variable = string expression	Moves data from memory to a random file buffer (left-justified).

OPEN filespec FOR "file mode 1" AS # file number LEN reclen	Opens a file or device.
OPEN "file mode 2", #, filespec, reclen	Alternative forms of OPEN.
OPEN. "file mode 2", #, file number, path, reclen	
PRINT # file number, list of expressions	Writes data to a sequential disk file.
PRINT # file number, USING "string exp." list of exps	Writes data to a sequential file.
PUT # file number, record number	Writes a record from a random buffer to a random disk file.
RSET string variable = string expression	Moves data from memory to a random file buffer (right-justifies).
WRITE # file number, list of expressions	Writes data to a sequential disk file.

Input/Output (Terminal) Statements

INPUT "prompt string"; list of variables	Allows input from the terminal during program execution.
LINE INPUT "prompt string"; string variable	Inputs an entire line to a string variable.
LPRINT list of expressions	Prints data on the line printer.

LPRINT USING string expression; list of expressions	Prints data on the line printer.
PRINT list of expressions	Outputs data at the terminal.
PRINT USING "string expression"; list of expressions	Prints strings or numbers using a specified format.
WRITE list of expression	Outputs data at the terminal.

Music Statements

BEEP	Sounds the speaker at approximately 1000 Hz for 1/4 second.
ON PLAY (n) GOSUB line number	Allows continuous background music to be played during program execution.
PLAY string expression	Plays melody indicated by a character string.
SOUND frequency duration	Generates sound through the speaker.

Non-I/O Statements

CALL variable name (argument list, argument list)	Calls the machine language subroutine.
CHAIN "filename", line number	Calls a program and passes variables to that program from the current program.

COMMON list of variables	Passes variables to a chained program.
DATA list of constants	Stores numeric and string constants.
DEF type range of letters	Declares variable types as integer, single precision, double precision, or string.
DEF FN name (parameter list) = function definition	Defines and names a function.
DEF SEG = address	Defines the current segment of storage.
DEF USR digit = integer expression	Specifies the starting address of an assembly language subroutine.
DIM list of subscripted variables	Specifies the maximum values for array variable subscripts and allocates storage accordingly.
END	Terminates program execution, closes all files, and returns to command level.
ERASE list of array variables	Eliminates arrays in a program.
FOR variable = x TO y STEP NEXT variable, variable	Executes a series of statements to be performed in a loop a given number of times.

Appendix I

GOSUB line number... RETURN	Branches to a subroutine.
GOTO line number	Branches unconditionally out of the normal program sequence to a specified line number.
IF expression THEN statements ELSE statements	Makes a decision regarding program flow based on the result returned by an expression.
LET variable = expression	Assigns the value of an expression to a variable.
MID$ (string expression 1, n,m) = string expression 2	Replaces a portion of one string with another string.
NEXT variable	Specifies the end of a FOR loop.
ON expression GOSUB list of line numbers	Branches to one of several specified subroutines.
ON expression GOTO list of line numbers	Branches to one of several specified line numbers.
ON TIMER (seconds) GOSUB line number	Transfers control to a specific program line after a specified period of time.
OPTION BASE n	Declares the minimum value for array subscripts.

RANDOMIZE expression	Reseeds the random number generator.
RANDOMIZE TIMER	Changes the sequence of random numbers without a prompt.
REM remark	Allows insertion of explanatory remarks in a program.
READ list of variables	Reads values from a DATA statement and assigns them to variables.
RESTORE line number	Rereads DATA statements from a specified line.
RETURN line number	Returns from a subroutine.
STOP	Terminates program execution and returns to command level.
SWAP variable, variable	Exchanges the value of two variables.
WEND	Specifies the end of a WHILE loop.
WHILE expression loop statements	Executes a series of statements in a loop as long as a given condition is true.

Appendix I

Port Statements

OUT port, data	Sends a byte to a machine output port.
POKE port, data	Writes a byte into a memory location.
WAIT port, n, m	Suspends program execution while monitoring the status of a machine input port.

Screen Statements

CIRCLE (xcenter, ycenter), radius, color, start, end, aspect	Draws a circle with a center and radius.
CLS screen identifier	Erases the currently active screen page.
COLOR foreground, background, border	Sets colors on the text screen.
COLOR background, palette, foreground	Sets colors in the graphic mode.
COLOR foreground, background, border, foreground	Sets colors in the superimposed mode.
DRAW "command string"	Draws figures on the screen.
GET (x1, y1)-(x2, y2) array name	Reads points from an area of the screen.

LINE (x1, y1)-(x2, y2), color, BF, style	Draws straight lines and rectangles in graphic mode.
LOCATE row, column, cursor start, stop	Moves the cursor to the specified position on the active screen.
MID$ (string expression 1, n, m)=string expression 2	Replaces a portion of one string with another string.
SCREEN mode, burst, apage, vpage	Specifies monitor attributes.
PAINT (x, y) paint, border	Fills in graphic figure with the specified color.
PALETTE register number, color	Assigns a color to a color register.
PALETTE USING array variable	Assigns 16 colors to all color registers.
PRESET (x, y), color	Draws a dot at the specified position on the screen in the background color.
PSET (x, y), color	Draws a dot at the specified position on the screen in the foreground color.
PUT (x, y), array name, operation	Outputs graphic patterns in the specified position on the screen.

VIEW SCREEN (x1, y1)–(x2, y2), color, boundary	Defines viewports onto which window contents are mapped.
WIDTH "device", size	Sets the width in characters for the screen (or line printer).
WINDOW SCREEN (x1, y1)–(x2, y2)	Redefines screen coordinates so the logical screen can exceed the boundaries of the physical screen.

Functions and Variables

Arithmetic Functions

ABS (x)	Returns the absolute value of the expression x.
ATN (x)	Returns the arctangent of x in radians.
CDBL (x)	Converts x to a double precision number.
CINT (x)	Converts x to an integer by rounding the fractional portion.
COS (x)	Returns the cosine of x in radians.
CSNG (x)	Converts x to a single precision number.
EXP (x)	Returns e to the power of x.

FIX (x)	Returns the truncated integer part of x.
INT (x)	Returns the largest integer less than or equal to x.
LOG (x)	Returns the natural logarithm of x.
RND (x)	Returns a random number between 0 and 1.
SGN (x)	Returns a value depending on the sign of x.
SIN (x)	Returns the sine of x in radians.
SQR (x)	Returns the square root of x.
TAN (x)	Returns the tangent of x in radians.

Child Process Handling Function

ENVIRON$ (parameter)	Retrieves the specified environment string from the BASIC environment string table.

Device Driver Handling Function

IOCTL$ (#n)	Reads a control data string from a character device driver.

Appendix I

Device Error Handling Functions

ERDEV	Returns the error number corresponding to the device error.
ERDEV$	Returns the name of the device for which the error is reported.

File Status Functions

EOF file number	Tests to see if the file specified has ended.
LOC file number	Returns the present location in the file.
LOF file number	Returns the number of bytes allocated to the file.

Input Functions

INKEY$	Returns a 1-character string containing a character from the terminal or a null string.
INPUT$ (x, # y)	Returns a string of x characters read from the terminal or from file y.

String Handling Functions

ASC (X$)	Returns a numeric value that is the ASCII code of the first character of string X$.
CHR$ (x)	Returns a string whose one element has ASCII code x.

CVI (2-byte string), CVS (4-byte string), CVD (8-byte string)	Converts a string value to a numeric value.
HEX$ (X)	Returns a string that represents the hexadecimal value of the decimal argument.
INSTR (1, X$, Y$)	Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found.
LEFT$ (X$, I)	Returns a string comprised of the leftmost I characters of X$.
LEN (X$)	Returns the number of characters in X$.
MID$ (X$, I, J)	Returns a string of length J characters from X$ (beginning with I).
MKI$ (integer expression), MKS$, (single precision expression), MKD$ (double precision expression	Converts numeric values to string values.
OCT$ (x)	Returns a string that represents the octal value of the decimal argument.
RIGHT$ (X$, I)	Returns the rightmost I character of string X$.

SPACE$ (x)	Returns a string of spaces the length of x.
STR$ (x)	Returns a string representation the value of x.
STRING$ (I, X$)	Returns a string of length I whose characters all have the first character of X$.
STRING$ (I, J)	Returns a string of length I whose characters all have ASCII code J.
VAL (X$)	Returns the numeric value of string X$.
VARPTR$ (variable)	Returns the address of a variable in character form.

Miscellaneous Functions

CSRLIN	Returns the current line (or row) position of the cursor.
DATE$ = stringexpression	Sets the date in calendar form.
String expressions = DATE$	Returns the current date in calendar form.
U = ERR	Returns the error code for the last error.
U = ERL	Returns the line number associated with the last error.

FRE (x)	Returns the number of bytes in memory not being used by BASIC.
INP (x)	Returns the byte read from port x.
LPOS (x)	Returns the current position of the line printer printhead within the line priner buffer.
PEEK (x)	Returns the byte (decimal integer) read from memory location x.
PLAY (n)	Reports the number of notes currently stored in the music background buffer.
V=PMAP (x, n)	Maps physical coordinates to world coordinates (and vice versa).
POINT (n)	Returns the current value of the current x or y coordinate.
POINT (x, y)	Returns the color of a pixel on the screen.
POS (l)	Returns the current cursor position.
X=SCREEN (row, col, z)	Returns the ASCII code (0 to 255) for the character at a specified row (line) and column of the screen.

Appendix I

SPC (x)	Prints x blanks on the screen.
TAB (x)	Spaces to position x on the screen.
TIME$ = string expression	Sets the time in character from.
String expression = TIME$	Returns the present time in character form.
x = TIMER	Returns the number of seconds that have elapsed since midnight or system reset.
USR digit (x)	Calls the user's assembly language subroutine to the argument x.
VARPTR (variable name)	Returns the address of the first byte of data identified with variable name.
VARPTR (# file number)	Returns the starting address of the file control block assigned to file number.

Appendix J. Key Scan Codes

Function Keys

Key	Hexadecimal Scan Code
F1	3B
F2	3C
F3	3D
F4	3E
F5	3F
F6	40
F7	41
F8	42
F9	43
F10	44

Special Function Keys

Key	Hexadecimal Scan Code
Esc	01
Ctrl	1D
Alt	3B

Appendix J

Alphabetic Keys

Key	Hexadecimal Scan Code	Key	Hexadecimal Scan Code
A	1E	N	31
B	30	O	18
C	2E	P	19
D	20	Q	10
E	12	R	13
F	21	S	1F
G	22	T	14
H	23	U	16
I	17	V	2F
J	24	W	11
K	25	X	2D
L	26	Y	15
M	32	Z	2C

Numeric Keys

Key	Hexadecimal Scan Code
1 !	02
2 @	03
3 #	04
4 $	05
5 %	06
6 ^	07
7 &	08
8 *	09
9 (	0A
0)	0B

Numeric Keypad

Key	Hexadecimal Scan Code
1 End	4F
2 ↓	50
3 PgDn	51
4 <-	4B
5	4C
6 ->	4D
7 Home	47
8 ↑	48
9 PgUp	49
0 Ins	52
. Del	53
-	4A
+	4E
PrtSc *	37

Punctuation Keys

Key		Hexadecimal Scan Code
\	|	2B
-	_	0C
=	+	0D
[	{	1A
]	}	1B
‘	~	29
;	:	27
’	”	28
,	<	33
.	>	34
/	?	35

Appendix J

Cursor Control Keys

Key	Hexadecimal Scan Code
Tab	0F
Back Space	0E
Space Bar	39

Control Keys

Key	Hexadecimal Scan Code
Left Shift	2A
Num Lock	45
Scroll Lock/Break	46
Return	1C
Right Shift	36
Caps Lock	3A
Enter	1C

Index

Index

Index

Index

Reader Comment Form

Your comments and suggestions help us to improve our books. Please take a moment to fill out and mail this postage-paid card, so that we may better meet your needs.

Please give specific examples in the Remarks section.

Did this book help you learn to use your computer? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

Is the information accurate? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

Is the information easy to understand? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

Is the information easy to find? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

Does the index contain the topics you looked for? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

Does the glossary adequately explain the words you looked for? ☐ Always ☐ Usually ☐ Sometimes ☐ Not at all

How would you rate this book? ☐ Good ☐ Fair ☐ Poor

What is your level of computer experience? ☐ Much ☐ Some ☐ None

What is your main use of your personal computer? ☐ Mainframe Workstation ☐ Word Processing ☐ Small Business ☐ Home Use

Remarks:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 2540 SALT LAKE CITY, UTAH

SPERRY PERSONAL COMPUTER
USER PUBLICATIONS
322 N. SPERRY WAY
SALT LAKE CITY, UT. 84116

FOLD

TAPE DO NOT STAPLE TAPE


SPERRY