Part II/ BASIC For TRSDOS Version 6 Reference Manual

Introduction

This part of the manual is about the BASIC language. BASIC for TRSDOS Version 6 is an "interpreter." When you run a program, it executes each statement one at a time. This makes it quick and easy to use. It also allows you to take advantage of many of TRSDOS Version 6's features, such as:

- Faster running programs
- Better graphics capabilities
- More print positions on the screen

About this Manual

This is a reference manual, not a tutorial. We assume you already know BASIC and are using this manual to quickly find the information you need.

Section III — Operations. This section shows how to load BASIC. It also demonstrates how to write, run and save a BASIC program on disk.

Section IV — The BASIC Language. This section includes a definition for each of BASIC's keywords (statements and functions) in alphabetical order. In addition, it shows how to write a program to store data on disk.

IMPORTANT NOTE: If you have read "Getting Started with TRS-80 BASIC", you need to know the differences between TRSDOS Version 1 and TRSDOS Version 6 BASIC. Appendix E shows these differences. These differences will often prevent a BASIC program written for TRSDOS Version 1 from running under TRSDOS Version 6, unless the program is modified. You also need to know how to use "disk files." This is explained in Chapter 5.

Notations

CAPITALS	material which must be entered exactly as it appears.
italics	words, letters, characters or values you must supply from a set of acceptable entries.
. . . (ellipsis)	items preceding the ellipsis may be repeated.
X'NNNN'	NNNN is a hexadecimal number.
O'NNNNN'	NNNNN is an octal number.
(KEYNAME)	one of the keys from your keyboard.

ƀ	a blank space character (ASCII code 32). For example, in BASICƀƀPROG there are two spaces between BASIC and PROG.

Terms

buffer	a number between 1 and 15. This refers to an area in memory that BASIC uses to create and access a disk file. Once you use a buffer to create a file, you cannot use it to create or access any other files; you must first close the file. You may only access an open file with the buffer used to open it.
[parameters]	information you supply to specify how a command is to operate. Parameters enclosed in brackets are optional.
[expressions]	values you supply for a function to evaluate. Expressions enclosed in brackets are optional.
syntax	a command with its parameter(s), or a function with its argument(s). This shows the format to use for entering a keyword in a program line.

Terms Used in Chapter 7 for Brevity:

line	a numeric value that identifies a BASIC program line. Each line has a number between Ø and 65529.
integer	any integer expression. It may consist of an integer, or several integers joined by operators. Integers are whole numbers between −32768 and 32767.
string	any string expression. It may consist of a string, or several strings joined by operators. A string is a sequence of characters which is to be taken verbatim.
number	any numeric expression. It may consist of a number, or several numbers joined by operators.
dummy number or dummy string	a number (or string) used as a parameter to meet syntactic requirements, but whose value is insignificant.

Part II is organized this way:

Section III. Operations

Chapter 1. Sample Session

Chapter 2. Command Mode
Execution Mode

Chapter 3. Line Edit Mode

Section IV. The BASIC Language

Chapter 4. BASIC Concepts

Chapter 5. Disk Files

Chapter 6. Introduction to BASIC Statements and Functions

Chapter 7. BASIC Statements and Functions

Section III/ Operations

Chapter 1/ Sample Session

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

The main steps in running a program are:

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

Loading BASIC

After you power up your system and install the diskette, the TRSDOS Version 6 start-up logo is displayed. Then, the following prompt appears: Date?

To answer this prompt, type today's date in this format: DD/MM/YY; then press (ENTER). For example, for December 1, 1983, type:

```
12/01/83 (ENTER)
```

The computer converts these numbers to: Thu, Dec 1, 1983 and displays the message "TRSDOS Ready". This indicates that you are at the Operating System level. To load BASIC into the system, type:

```
BASIC (ENTER)
```

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

You may now begin using BASIC.

Options for Loading BASIC

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

BASIC [program] ([F=number of files] [,M=highest memory location])

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

F= specifies the maximum number of data files that may be open at any one time (from 0-15). If you omit this option, the number of files defaults to three. Each file you specify uses 564 bytes of memory.

M= specifies the highest memory location for BASIC to use. Omit this option unless you are going to call assembly-language subroutines. (In that case, you may want to set the amount of memory well below the high-memory modules of TRSDOS.) If you

omit this option, the system allocates all memory up to the HIGH$ marker to BASIC. HIGH$ can be adjusted through the MEMORY library command. See the TRSDOS Reference Manual for more details.

Examples

```
TRSDOS Ready
BASIC PAYROLL (F=5) (ENTER)
```

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

```
TRSDOS Ready
BASIC (M=45056) (ENTER)
```

initializes BASIC; allows three data files to be open; sets the highest memory location to be used by BASIC at 45056.

```
TRSDOS Ready
BASIC (M=32768, F=6) (ENTER)
```

initializes BASIC; sets the highest memory location at 32768; allows six data files to be open. Notice that the sequence in which the M= and F= options are specified is irrelevant.

```
TRSDOS Ready
BASIC
```

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

Typing the Program

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

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

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

For example, suppose you had typed:

```
15 B$="THE VERCHANT OF VENICE"
```

To correct line 15, re-type it:

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

Then type:

```
RUN (ENTER)
```

Your screen should display:

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

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

NOTE: BASIC "reads" your program lines in numerical order. It doesn't matter if you entered line 15 after line 20; it will still read and execute 15 before "looking" at 20.

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

Saving the Program on Disk

You can save any of your BASIC programs on disk. To do this, you assign it a "filespec".

For example, if you wanted to save the program we just wrote, you could assign it the filespec "AUTHOR". Type the following command:

```
SAVE "AUTHOR" (ENTER)
```

It takes a few seconds for the computer to find a place on disk to store our program. When this process is completed, it displays Ready. The program is now saved on disk.

IMPORTANT NOTE: A filespec can have a maximum of eight alphanumeric characters. It can also have an optional extension, up to three characters long. A slash / must be included between the filespec and the extension. The first character of both the filespec and the extension must be a letter.

Example

```
SAVE "AUTHOR/WIL" (ENTER)
```

You may also add a drive number to your filespec by typing a colon : and the drive number.

Example

```
SAVE "AUTHOR:1" (ENTER)
```

tells the computer to save "AUTHOR" on the disk in Drive 1. Otherwise, the computer assumes you to save it on the first available drive. If you do specify a disk drive, make sure you have a disk in that drive.

Loading the Program

If, after writing or running other programs, you wanted to go back and use this program again, you must "load" it back into memory. To do this, type: LOAD "filespec", R

Example

```
LOAD "AUTHOR", R (ENTER)
```

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

Another way to load and run a program is to type: RUN "filespec". RUN automatically loads and runs the program specified by "filespec".

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

Chapter 2/ Command And Execution Modes

This chapter describes BASIC's command and execution modes. The command mode is for typing in program lines and immediate lines. The execution mode is for executing programs and immediate lines.

Command Mode

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

```
Ready
```

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

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

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

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

Interpretation of a Line

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

For example, if you type:

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

BASIC takes this as an immediate line.

But if you type:

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

BASIC takes this as a program line.

Immediate Lines

An immediate line consists of one or more statements separated by colons. The line is executed as soon as you press (ENTER). For example:

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

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

Program Lines

A program line consists of a line number in the range 0 to 65529, followed by one or more statements separated by colons. When you press (ENTER), the line is stored in memory, along with any other lines you have entered this way. The program is not executed until you type RUN or another execute command. For example:

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

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

```
RUN (ENTER)
```

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

Special Keys in the Command Mode

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

Execution Mode

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

Special Keys in the Execution Mode

(SHIFT)(@)	Pauses execution. Press any other key (except (BREAK) to continue.

(BREAK)	Terminates execution and returns you to command mode.
(ENTER)	Interprets data entered from the keyboard as a response to the INPUT statement.

Chapter 3/ Line Edit Mode

This mode enables you to "debug" (correct) programs quickly and efficiently. It allows you to correct a program line without having to re-type the entire line.

If your computer encounters a syntax error while executing a program, it automatically puts you in the "line edit mode." The display shows:

```
Syntax error in line number
Ready
line number
```

(line number is the program line in which the error occurred.) In this case, you are ready to use the edit mode commands and subcommands described later in this Chapter.

However, if you wish to activate the line editor yourself (because you have noticed a mistake or wish to make a change in a long program line), type:

```
EDIT line number (ENTER)
```

This lets you edit the specified line number. (If the line number you specify has not been used, an "Undefined line number" error occurs. If you do not have a space after the word EDIT, a "Syntax error" occurs.)

You may also type:

```
EDIT . (ENTER)
```

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

For example, type the following line and press (ENTER). (To type the exponent sign ^, press (CLEAR)(;)).

```
100 FOR I = 1 TO 10 STEP .5:  PRINT I, I^2, I^3:
    NEXT
```

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

Now type EDIT 100 and press (ENTER). The computer displays:

```
100
```

This starts the editor. You may now begin editing line 100.

Special Keys in the Edit Mode

(ENTER)

Pressing (ENTER) in the edit mode records all the changes you made in the current line and returns you to the command mode.

Space bar

Pressing the space bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the computer in the edit mode so the display shows:

```
100
```

Now press the space bar. The cursor moves mover one space and the first character of the program line is displayed. If this character was a blank, then a blank is displayed. Press the space bar again until you reach the first non-blank character:

```
100 F
```

is displayed. To move over more than one space at a time, type the desired number of spaces first, then press the space bar. For example, type 6 and press the space bar. The display should show something like this (depending on how many blanks you inserted in the line):

```
100 FOR I =
```

Now type 8 and press the space bar. The cursor moves over eight spaces to the right, and eight more characters are displayed.

```
100 FOR I = 1 TO 10
```

L (List Line)

displays the remainder of the program line (unless the computer is under one of the insert subcommands listed below). The cursor drops down to the next line of the display, reprints the current line number, and moves to the first position of the line.

For example, when the display shows

```
100
```

press L (without pressing (ENTER)). Line 100 is displayed:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I,' I^2, I^3:
    NEXT 100
```

This lets you look at the line in its current form while you're doing the editing.

Insert Subcommand Mode

The insert subcommand mode allows you to add material to a line while editing it. The three keys you can use to enter this subcommand mode are X, I and H.

X (Extend Line)

Displays the rest of the current line. Typing (X) also moves the cursor to the end of the line and puts the computer in the insert subcommand mode. This enables you to add material to the end of the line.

For example, using line 100, when the display shows

```
100
```

press (X) (without pressing (ENTER)) and the entire line is displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2,
     I^3: NEXT
```

We can now add another statement to the line, or delete material from the line by using the (←) key. For example, type

```
: PRINT "DONE"
(ENTER)
```

at the end of the line. If you typed:

```
LIST 100
```

the display should show something like this:

```
100 FOR I = 1 TO 10 STEP .5: PRINT I, I^2,
     I^3: NEXT: PRINT "DONE"
```

NOTE: If you want to continue editing the line, press (SHIFT)(↑) to get out of the insert subcommand mode.

I (Insert)

Inserts material beginning at the current cursor position on the line.

For example, type

```
EDIT 100
(ENTER)
```

then use the space bar to move over to the decimal point in line 100. The display shows:

```
100 FOR I = 1 TO 10 STEP .
```

Suppose you want to change the increment from .5 to .25. Press the (I) key (don't press (ENTER)). The computer lets you insert material at

the current position. Type 2 now, and the display shows:

```
100 FOR I = 1 TO 10 STEP .2
```

You have made the necessary change, so press (SHIFT)(↑) to escape from the insert subcommand. Now press (L) to display the remainder of the line and move the cursor back to the beginning of the line:

```
100 FOR I = 1 TO 10 STEP .25: PRINT I, I^2,
    I^3: NEXT: PRINT "DONE"
100
```

NOTE: You can also exit the insert subcommand and save all changes by pressing (ENTER). This returns you to command mode.

H (Hack and Insert)

Deletes the remainder of a line and lets you insert material at the current cursor position.

For example, using line 100, enter the edit mode and space over until just before the PRINT "DONE" statement. Suppose you wanted to delete this statement and insert an END statement. The display shows:

```
100 FOR I = 1 TO 10  STEP .25:  PRINT I,  I^2,
    I^3:  NEXT:
```

Press (H), then type END and press (ENTER). List the line:

```
100 FOR I = 1 TO 10  STEP .25:  PRINT I,  I^2,
    I^3:  NEXT:  END
```

should be displayed.

NOTE: To continue editing the line, press (SHIFT)(↑) to get out of the insert subcommand mode.

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes already made.

For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first press (SHIFT)(↑) (to escape from any subcommand you may be executing); then press (A). The cursor drops down to the next line, displays the line number and moves to the first character position.

E (Save Changes and Exit)

Ends editing and saves all changes made. You must be in edit mode, not executing any subcommand, when you press (E) to end editing.

Q (Cancel and Exit)

Ends editing and cancels all changes made in the current editing session. If you've decided not to change the line, type (Q) to cancel changes and leave the edit mode.

If a syntax error is detected during program execution, BASIC starts the editor. To examine variable values, you must press Q before typing any other command.

nD (Delete)

Deletes the specified number n of characters to the right of the cursor. The deleted characters appear enclosed in exclamation points.

For example, using line 100, space over to just before the PRINT statement:

```
100 FOR I = 1 TO 10 STEP .25:
```

Now type 19D. This tells the computer to delete 19 characters to the right of the cursor. The display should show something like this:

```
100 FOR I = 1 TO 10 STEP .25: \PRINT I, I^2,
    I^3:\
```

When you list the complete line, you will see that everything from the PRINT to the next statement has been deleted.

nC (Change)

Lets you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the computer assumes you want to change one character. When you have entered n number of characters, the computer returns you to the edit mode (so you're not in the nC subcommand).

For example, using line 100, suppose you want to change the final value of the FOR NEXT loop, from "10" to "15". In the edit mode, space over to just before the "0" in "10".

```
100 FOR I = 1 TO 1
```

Now press (C). The computer assumes you want to change just one character. Press (5), then press (L). When you list the line, you will see that the change has been made.

```
100 FOR I = 1 TO 15 STEP .25: NEXT: END
```

would be the current line if you've followed the editing sequence in this chapter.

nSc (Search)

Searches for the nth occurrence of the character c, and moves the cursor to that position. If you don't specify a value for n, the computer searches for the first occurrence of the specified character. If character c is not found, cursor goes to the end of the line.

NOTE: The computer only searches through characters to the right of the cursor.

For example, using the current form of line 100 type EDIT 100 (ENTER), then press (2)(S)(:). This tells the computer to search for the second occurrence of the colon character. The display should show:

```
100 FOR I = 1 TO 15 STEP .25: NEXT
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the insert subcommand, then type the variable name, I. That's all you want to insert, so press (SHIFT)(↑) to escape from the insert subcommand mode. The next time you list the line, it should appear as:

```
100 FOR I = 1 TO 15 STEP .25: NEXT I: END
```

nKc (Search and "Kill")

Deletes all characters up to the nth occurrence of character c, and moves the cursor to that position.

For example, using the current version of line 100, suppose we wanted to delete the entire line up to the END statement. Type EDIT 100 (ENTER), then type (2)(K)(:). This tells the computer to delete all characters up to the 2nd occurrence of the colon.

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\
```

should be displayed. The second colon still needs to be deleted, so type D. The display now shows:

```
100 \FOR I = 1 TO 15 STEP .25: NEXT I\\:\
```

Press (ENTER) and type LIST 100 (ENTER)

Line 100 should look something like this:

```
100 END
```

n (←)

Moves the cursor to the left by n spaces. If no number n is given, the cursor moves back one space. When the cursor backspaces, all characters in its path are erased from the display, but they are not deleted from the program. Use the space bar to advance the cursor forward and re-display the erased characters.

Section IV/ The BASIC Language

Chapter 4/ BASIC Concepts

This chapter explains how to use the full power of BASIC for TRSDOS Version 6. This information can help programmers build powerful and efficient programs. If you are still something of a novice, you might want to skip this chapter for now, keeping in mind that the information is here when you need it.

The chapter is divided into four sections:

A. Overview — Elements of a Program. This section defines many of the terms we will be using in the chapter.

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

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

D. How to Construct an Expression. This topic can help you in constructing powerful statements instead of using many short ones.

A- Overview: Elements of a Program

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

We will refer to these terms during the rest of this chapter.

Program

A program is made up of one or more numbered lines. Each line contains one or more BASIC statements. BASIC allows line numbers from 0 to 65529 inclusive. You may include up to 255 characters per line, including the line number. You may also have two or more statements to a line, separated by colons.

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

Here is a sample program:

Line number — **BASIC statement** — **Colon between statements** — **BASIC statement**

```
100 CLS: PRINT "NORMAL MODE..."
110 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
120 FOR I = 1 TO 1000: NEXT I
130 CLS: PRINT CHR$(23); "DOUBLE-SIZE MODE..."
140 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
150 END
```

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

Statements

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

```
GOTO 100
```

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

```
END
```

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

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

```
PRINT"SEPTEMBER REPORT"
```

the data is SEPTEMBER REPORT. The statement instructs the computer to print the data inside quotes.

Expressions

An expression is actually a general term for data. There are four types of expressions:

1. Numeric expressions, which are composed of numeric data.

Examples:

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

2. String expressions, which are composed of character data.

Examples:

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

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

Examples:

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

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

Examples:

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

Functions

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

These are some of BASIC's functions:

```
INT
ABS
STRING$
```

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

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

B- How BASIC Handles Data

BASIC for TRSDOS Version 6 offers several different methods of handling your data. Using these methods properly can greatly improve the efficiency of your program. In this section we discuss:

Ways of Representing Data
- Constants
- Variables

How BASIC Stores Data
- Numeric (integer, single precision, double precision)
- String

How BASIC Classifies Constants
How BASIC Classifies Variables
How BASIC Converts Data

Ways of Representing Data

BASIC recognizes data in two forms: directly (as constants), or by reference to a memory location (as variables).

Constants

All data is input into a program as "constants" — values which are not subject to change. For example, the statement:

```
PRINT "1 PLUS 1 EQUALS"; 2
```

contains one string constant (1 PLUS 1 EQUALS), and one numeric constant (2).

In these examples, the constants "input" to the PRINT statement. They tell PRINT what data to print on the display.

These are more examples of constants:

3.14159	"L.O.SMITH"
1.775E+3	"0123456789ABCDEF"
"NAME TITLE"	−123.45E−8
57	"AGE"

Variables

A variable is a place in memory where data is stored. Unlike a constant, a variable's value can change. This allows you to write programs dealing with changing quantities. For example, in the statement:

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

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

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

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

Variables can also store numeric values. For example:

```
A = 134
```

Variable Names

In BASIC, variables are represented by names. Variable names must begin with a letter, A through Z. This letter may be followed by one or more characters (digits or letters).

For example:

AM   A   A1   BALANCE   EMPLOYEE2

are all valid and distinct variable names.

Variable names may be up to 40 characters long. All characters are significant in BASIC.

Reserved Words

Certain combinations of letters are reserved as BASIC keywords and operator names. These combinations cannot be used as variable names. For example:

OR   LEN   OPTION

cannot be used as variable names. However, they may be embedded in a variable name. For example, OPTIONS is a valid variable name.

TRSDOS Version 6 requires that all reserved words be delimited. This means that you must leave a blank space between a reserved word and any variables, constants or other reserved words. See Appendix F for a list of BASIC's reserved words.

Simple and Subscripted Variables

Variables may also be "subscripted" so that an entire list of data can be stored under one variable name. This method of data storage is called an *array*. For example, an array named A may contain these elements (subscripted variables):

A(0)   A(1)   A(2)   A(3)   A(4)

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

```
A(0) = 5.3
A(1) = 7.2
A(2) = 8.3
A(3) = 6.8
A(4) = 3.7
```

In this example, array A is a one-dimensional array, since each element contains only one subscript. An array may also be two-dimensional, with each element containing two subscripts. For example, a two-dimensional array named X could contain these elements:

X(0,0) = 8.6   X(0,1) = 3.5
X(1,0) = 7.3   X(1,1) = 32.6

With BASIC, you may have as many dimensions in your array as your program space allows. Here is an example of a three-dimensional array named L which contains these eight elements:

L(0,0,0) = 35233 L(0,1,0) = 96522
L(0,0,1) = 52000 L(0,1,1) = 10255

L(1,0,0) = 33333 L(1,1,0) = 96253
L(1,0,1) = 53853 L(1,1,1) = 79654

BASIC assumes that all arrays contain 11 elements in each dimension. If you want more elements you must use the DIM statement at the beginning of your program to dimension the array.

For example, to dimension array L, put this line at the beginning of the program:

```
DIM L(1,1,1)
```

to allow room for two elements in the first dimension; two in the second, and two in the third for a total of 2 * 2 * 2 = 8 elements.

How BASIC Stores Data

The way BASIC stores data determines the amount of memory it consumes and the speed in which BASIC can process it.

Numeric Data

You may get BASIC to store all numbers in your program as either integer, single precision, or double precision. In deciding how to get BASIC to store your numeric data, remember the tradeoffs. Integers are the most efficient and the least precise. Double precision is the most precise and least efficient.

Integers
(Fastest in Computations, Limited in Range)

To be stored as an integer, a number must be whole and in the range of −32768 to 32767. An integer value requires two bytes of memory for storage. Arithmetic operations are faster when both operands are integers.

For example:

1   3200   −2   500   −12345

can all be stored as integers.

Single Precision
(General Purpose, Full Numeric Range)

Single-precision numbers can include up to seven significant digits, and can represent normalized values* with exponents up to 38, i.e., numbers in the range:

$$[-1 \times 10^{38}, -1 \times 10^{-38}]\ [1 \times 10^{38}, 1 \times 10^{-38}]$$

If a number is raised to a power greater than 38, an "Overflow" error occurs. If it is raised to a power lower than −38, no errors are generated and program execution continues.

A single-precision value requires four bytes of memory for storage. BASIC assumes a number is single precision if you do not specify the level of precision.

* In this manual, normalized value is one in which exactly one digit appears to the left of the decimal point. For example, 12.3 expressed in normalized form is 1.23×10.

For example:

10.001 −200034 1.774E6 6.024E−23 123.4567

can all be stored as single-precision values. But even though BASIC stores a number with up to seven digits of precision, when printing it, only six digits are shown.

NOTE: When used in a decimal number, the symbol E stands for "single-precision times 10 to the power of . . ." Therefore 6.024E−23 represents the single-precision value:

$$6.024 \times 10^{-23}$$

Double Precision
(Maximum Precision, Slowest in Computations)

Double-precision numbers can include up to 16 significant digits, and can represent values in the same range as that for single-precision numbers. A double-precision value requires eight bytes of memory for storage. Arithmetic operations involving at least one double-precision number are slower than the same operations when all operands are single precision or integer.

For example:

1010234578
−8.7777651010
3.141592653589793
8.00100708D12

can all be stored as double-precision values.

NOTE: When used in a decimal number, the symbol D stands for "double precision times 10 to the power of . . ." Therefore

8.00100708D12 represents the value

$8.00100708 \times 10^{12}$

Strings

Strings (sequences of characters) are useful for storing non-numeric information such as names, addresses, or text. You may store ASCII characters, as well as any of the graphic and non-ASCII symbols, in a string. (A list of Character Codes is included in Appendix C).

For example, the data constant:

Jack Brown, Age 38

can be stored as a string of 18 characters. Each character (and blank) in the string is stored as an ASCII code, requiring one byte of storage.

BASIC would store the above string constant internally as:

Hex Code	4A	61	63	6B	20	42	72	6F	77	6E	2C	20	41	67	65	20	33	38
ASCII Character	J	a	c	k		B	r	o	w	n	,		A	g	e		3	8

A string can be up to 255 characters long. Strings with length zero are called "null" or "empty".

How BASIC Classifies Constants

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

Rule 1

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

For example:

"YES"
"3331 Waverly Way"
"1234567890"

are all classified as strings.

Rule 2

If the value is not in quotes, it is a number. (An exception to this rule is during data input by an operator, and in DATA lists. See INPUT, INKEY$, and DATA)

For example:

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

are all numeric data.

Rule 3

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

For example:

```
12350
-12
10012
```

are integer constants.

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

Rule 4

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

For example:

```
1234567
-1.23
1.3321
```

are all classified as single precision.

Rule 5

If the number contains more than seven digits, it is double precision. For example, these numbers:

```
12345678901234556
-1000000000000.1
2.777000321
```

are all classified as double precision.

Type Declaration Tags

You can override BASIC's normal typing criteria by adding the following "tags" at the end of the numeric constant:

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

```
A = 12.345678901234!
```

BASIC classifies the constant as single precision, and shortens it to seven digits. However, if you tell BASIC to print the value of A, only six digits are printed out:

```
12.3457
```

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

```
A = 1.2E5
```

stores the single-precision number 120000 in A.

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

```
PRINT 3#/7
```

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

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

```
A = 1.23456789D - 1
```

the double-precision constant has the value 0.123456789.

How BASIC Classifies Variables

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

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

AB AMOUNT XY L

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

```
LP = 1.2
```

BASIC classifies LP as a single-precision variable.

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

DEFINT - Defines variables as integer
DEFDBL - Defines variables as double-precision
DEFSTR - Defines variables as string
DEFSNG - Defines variables as single-precision. (Since BASIC classifies all variables as single precision initially

anyway, you would only need to use DEFSNG if one of the other DEF statements was used).

For example:

```
DEFSTR L
```

makes BASIC classify all variables which start with L as string variables. After this statement, the variables:

L   LP   LAST

can all hold string values only.

Type Declaration Tags

As with constants, you can always override the type of a variable name by adding a type declaration tag at the end. The four types of declaration tags for variables are:

% Integer
! Single precision
Double precision
$ String

For example:

I%   FT%   NUM%   COUNTER%

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

T!   RY!   QUAN!   PERCENT!

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

X#   RR#   PREV#   LSTNUM#

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

Q$ CA$ WRD$ ENTRY$

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

Note that any given variable name can represent four different variables. For example:

A5#   A5!   A5%   A5$

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

One further implication of type declaration: Any variable name used without a tag is equivalent to the same variable name used with one of the four tags. For example, after the statement:

```
DEFSTR C
```

the variable referenced by the name C1 is identical to the variable referenced by the name C1$.

How BASIC Converts Numeric Data

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

```
A% = 2.34
```

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

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

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

The conversion procedures are explained on the following pages.

Single or double precision to integer type

BASIC rounds the fractional portion of the number.

NOTE: The original value must be greater than or equal to −32768, and less than 32768.

Examples

```
A% = 32766.7
```

assigns A% the value 32767.

```
A% = 2.5D3
```

assigns A% the value 2500.

```
A% = -123.45678901234578
```

assigns A% the value −123.

```
A% = -32768.5
```

produces an Overflow Error (out of integer range).

Integer to single or double precision

No error is introduced. The converted value looks like the original value with zeros to the right of the decimal place.

Examples

```
A# = 32767
```

Stores 32767.000000000000 in A#.

```
A! = -1234
```

Stores −1234.000 in A!.

Double to single precision

This involves converting a number with up to 16 significant digits into a number with no more than seven digits. BASIC rounds the number to seven significant digits. Before printing it, BASIC rounds it off to six digits.

Examples

```
A! = 1.234567890124567
```

stores 1.234568 in A!. However, the statement:

```
PRINT A
```

displays the value 1.23457, because only six digits are displayed. The full seven digits are stored in memory.

```
A! = 1.3333333333333333
```

stores 1.333333 in A!.

Single to double precision

To make this conversion, BASIC simply adds trailing zeros to the single-precision number. If the original value has an exact binary representation in single-precision format, no error is introduced. For example:

```
A# = 1.5
```

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

However, for numbers which have no exact binary representation, an error is introduced when zeros are added. For example:

```
A# = 1.3
```

stores 1.299999952316284 in A#.

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

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

both store 1.3 in A#.

Here is a special technique for converting a single precision value to double precision, without introducing an error into the double-precision

value. It is useful when the single-precision value is stored in a variable.

Take the single-precision variable, convert it to a string with STR$, then convert the resultant string back into a number with VAL. That is, use:

VAL(STR$(single-precision variable))

For example, the following program:

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

prints a value of:

```
1.2999999523162842
```

Compare with this program:

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

which prints a value of:

```
1.3
```

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

Illegal Conversions

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

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

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

C- How BASIC Manipulates Data

You have many fast methods you may use to get BASIC to count, sort, test, and rearrange your data. These methods fall into two categories:

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

Operators

An operator is the single symbol or word which signifies some action to be taken on either one or two specified values referred to as operands.

In general, an operator is used like this:

operand-1	operator	operand-2
6	+	2

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

Operand-1 and -2 can be expressions.

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

operator	operand
–	5

The negative operator – acts on single operand 5 to produce the result negative 5.

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

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

Operators fall into four categories:

- Numeric
- String
- Relational
- Logical

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

Numeric Operators

Numeric Operators are used in numeric expressions. Their operands must always be numeric, and the results they produce is one numeric data item.

In the description below, we use the terms integer, single-precision, and double-precision operations. Integer operations involve two-byte operands, single-precision operations involve four-byte operands, and double-precision operations involve eight-byte operands. The more bytes involved, the slower the operation.

There are five different numeric operators. Two of them, sign + and sign –, are unary, that is, they have only one operand. A sign operator has no effect on the precision of its operand.

For example, in the statement:

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

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

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

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

These operators are, in order of precedence:

^	Exponentiation
*, /	Multiplication, Division
+, –	Addition, Subtraction

Exponentiation

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

NOTE: To enter the ^ operator, press (CLEAR)(;).

For example:

```
PRINT 6^.3
```

prints 6 to the .3 power.

Multiplication

The * operator is the symbol for multiplication. Once again, BASIC uses the precision of the more precise operand to perform the operation (the less precise operand is converted).

Examples:

```
PRINT 33 * 11%
```

integer multiplication is performed.

```
PRINT 33 * 11.1
```

single-precision multiplication is performed.

```
PRINT 12.345678901234567 * 11
```

double-precision multiplication is performed.

Division

The / symbol is used to indicate ordinary division. Both operands are converted to single precision or double precision, depending on their original precision:

- If either operand is double precision, then both are converted to double precision and eight-byte division is performed.
- If neither operand is double precision, then both are converted to single precision and four-byte division is performed.

Examples:

```
PRINT 3/4
```

single-precision division is performed.

```
PRINT 3.8/4
```

single-precision division is performed.

```
PRINT 3/1.2345678901234567
```

double-precision division is performed.

Addition

The + operator is the symbol for addition. The addition is done with the precision of the more precise operand (the less precise operand is converted).

For example, when one operand is integer type and the other is single precision, the integer is converted to single precision and four-byte addition is performed. When one operand is single precision and the other is double precision, the single-precision number is converted to double precision and eight-byte addition is performed.

Examples:

```
PRINT 2 + 3
```

integer addition is performed.

```
PRINT 3.1 + 3
```

single-precision addition is performed.

```
PRINT 1.2345678901234567 + 1
```

double-precision addition is performed.

Subtraction

The − operator is the symbol for subtraction. As with addition, the operation is done with the precision of the more precise operand (the less precise operand is converted).

Examples:

```
PRINT 33 - 11
```

integer subtraction is performed.

```
PRINT 33 - 11.1
```

single-precision subtraction is performed.

```
PRINT 12.345678901234567 - 11
```

double-precision subtraction is performed.

String Operator

BASIC has a string operator (+) which allows you to concatenate (link) two strings into one. This operator should be used as part of a string expression. The operands are both strings and the resulting value is one piece of string data.

The + operator links the string on the right of the sign to the string on the left. For example:

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

prints:

```
CATSLOVEMICE
```

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

Relational Operators

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

Numeric Relations

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

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

Examples of true relational expressions:

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

String Relations

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

<	Precedes
>	Follows
>< or <>	Does not have the same precedence
<=	Precedes or has the same precedence
>=	Follows or has the same precedence

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

NOTE: Appendix C contains a listing of ASCII codes for each character.

Examples of true relational expressions:

"A" < "B"

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

"CODE" < "COOL"

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

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

"TRAIL" < "TRAILER"

Leading and trailing blanks are significant. For example:

" A" < "A"

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

"Z-80" < "Z-80A"

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

How to Use Relational Expressions

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

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

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

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

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

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

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

However, you may also use relational expressions simply to return the true or false results of a test. For example:

```
PRINT 7 = 7
```

prints -1 since the relation tested is true.

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

prints 0 because the relation tested is false.

Logical Operators

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

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

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

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

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

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

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

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

Hierarchy of Operators

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

Parentheses

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

8 − (3 − 2)

is evaluated like this:

3 − 2 = 1
8 − 1 = 7

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

```
4 * (2 - (3 - 4))
```

is evaluated like this:

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

Order of Operations

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

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

For Numeric Operations:

```
( ) (Parentheses)
^ (Exponentiation)
+,- (Unary sign operands [not addition and subtraction])
*,/ (Multiplication and division)
+,- (Addition and subtraction)
<,>,=,<=,>=,< >
NOT
AND
OR
XOR
EQV
IMP
```

For String Operations:

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

For example, in the line:

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

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

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

or

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

Here's another example:

```
IF X + Ø  OR  Y > Ø  AND  Z = 1  THEN GOTO  255
```

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

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

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

Functions

A function is a built-in sequence of operations which BASIC performs on data. BASIC functions save you from having to write a BASIC routine, and they operate faster than a BASIC routine would.

Examples:

```
SQR (A + 6)
```

tells BASIC to compute the square root of (A + 6).

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

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

BASIC functions are described in more detail in Chapter 7.

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

D- How to Construct an Expression

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

- Simple
- Complex

as well as how to construct a function.

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

- Constants
- Variables
- Operators
- Functions

Expressions may be either simple or complex:

A **simple expression** consists of a single term: a constant, variable or function. If it is a numeric term, it may be preceded by an optional + or − sign, or by the logical operator NOT.

For example:

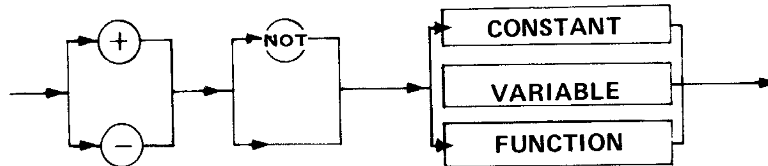
+A   3.3   −5   SQR(8)

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

A$ STRING$ (2Ø,A$)   "WORD"   "M"

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

Here's how a **simple expression** is formed:



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

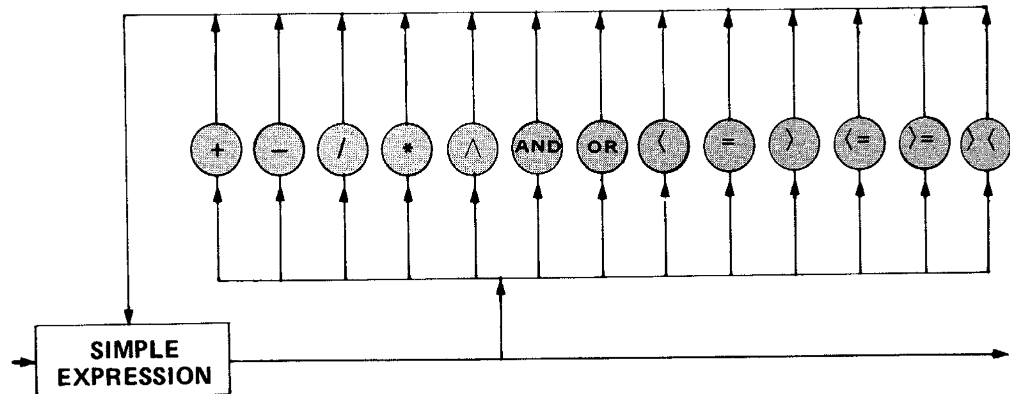
A-1   X+3.2−Y   1=1   A AND B   ABS(B)+LOG(2)

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

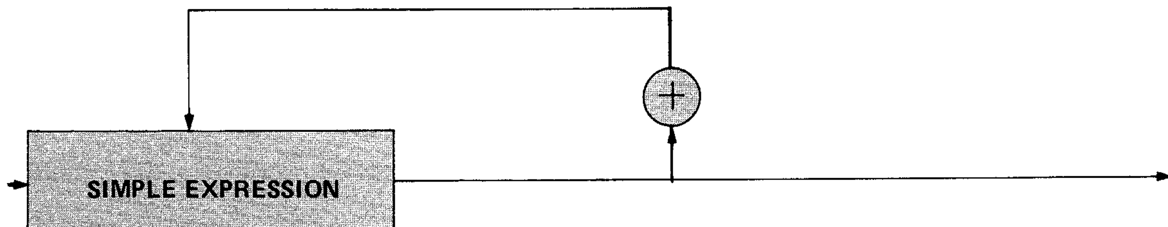
A$ + B$ "Z" + Z$ STRING$(1Ø, "A") + "M"

are all examples of complex string expressions.

This is how a **complex numeric expression** is formed:



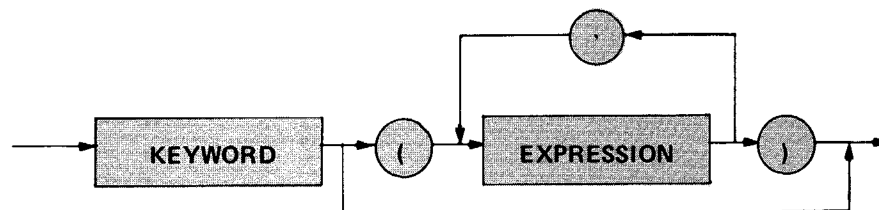
This is how a **complex string expression** is formed:



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

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

This is how a **function** is formed:



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

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

are all examples of functions.

Chapter 5/ Disk Files

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

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

With BASIC, you can create and access two types of disk files. The difference between these two types is that each is created in a different "mode." The mode you choose determines what kind of access you will have to the file: sequential access or direct access.

Sequential-Access Files

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

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

The statements and functions used with sequential files are:

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

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

Creating a Sequential-Access File

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

   Example

   ```
   OPEN "O", 1, "LIST/EMP"
   ```

   opens a sequential output file named LIST/EMP and gives buffer 1 access to this file.

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

Example

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

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

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

   Example

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

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

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

   Example

   ```
   CLOSE 1
   ```

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

   Sample Program

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

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

   Example

   ```
   SAVE "PAYROLL"
   ```

   would save the program under the name "PAYROLL".

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

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

   Example

   ```
   OPEN "I", 1, "LIST/EMP"
   ```

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

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

Examples

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

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

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

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

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

Sample Program

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

Updating a Sequential-Access File

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

```
OPEN "E", 1, "LIST/EMP"
```

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

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

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

NOTE: Read through the entire program first. If you encounter BASIC words (commands or functions) that are unfamiliar to you, refer to Chapter 7 for their definitions.

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

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

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

Once you have RUN this program, SAVE it.

Example

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

Direct-Access Files

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

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

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

The maximum number of logical records is 65,535. Each record may contain between 1 and 256 bytes.

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

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

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

Creating a Direct-Access File

1. To create the file, OPEN it for direct access in "D" mode ("R" may also be used. It stands for "random access", which is simply another name for direct access).

Example

```
OPEN, "D", 1, "LISTING", 32
```

opens the file named "LISTING", gives buffer 1 direct access to the file, and sets the record length to 32 bytes. (If the record length is omitted, the default is 256 bytes). Remember that data is passed to and from disk in records.

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

Example

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

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

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

Example

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

Note: RSET right justifies a string into the buffer. For example, RSET N$=X$.

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

```
PUT 1, CODE%
```

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

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

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

```
100 LSET P$ = TEL$
110 PUT 1, CODE%
120 GOTO 30
130 CLOSE 1
```

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

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

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

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

Accessing a Direct-Access File

1. OPEN the file in "D" mode ("R" can also be used).

   Example

   ```
   OPEN "D", 1,"FILE",32
   ```

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

   Example

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

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

   Example

   ```
   GET 1, CODE%
   ```

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

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

   Example

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

The program may now access the data in the buffer.

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

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

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

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

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

```
$0.00
```

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

Chapter 6/ Introduction To BASIC Statements And Functions

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

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

Format for Chapter 7

Keyword

Syntax *parameter(s)* or (*expression(s)*)

Brief definition of keyword.

Detailed definition of keyword.

Example(s)

Sample Program(s)

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

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

IMPORTANT NOTE: BASIC for TRSDOS Version 6 requires that keywords be delimited by spaces. This means that you must leave a space between a keyword and any variables, constants or other keywords. The only exceptions to this rule are characters which are shown as part of the syntax of the keyword.

For example, if you typed:

```
DELETE.
```

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

For a definition of the terms and notation used in Chapter 7, see page 2-4 of the Introduction.

Statements

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

```
100 STOP
```

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

Statements for assigning values to variables and defining memory space:

CLEAR	clears all variables, allocates memory and stack space.
COMMON	passes variables to a CHAINed program.
DATA	stores data in your program so that you may assign it to a variable.
DEFDBL	defines variables as double precision.
DEF FN	defines a function according to your specifications.
DEFINT	defines variables as integers.
DEFSNG	defines variables as single precision.
DEFSTR	defines variables as strings.
DEF USR	defines the entry point for USR routines.
DIM	dimensions an array.
ERASE	erases an array.
LET	assigns a value to a variable (the keyword LET may be omitted).
MID$	replaces a portion of a string.
OPTION BASE	declares the minimum value for array subscripts.
RANDOM	reseeds the random number generator.
READ	reads data stored in the DATA statement and assigns it to a variable.
RESTORE	restores the DATA pointer.
SWAP	exchanges the values of variables.

Statements for altering program sequence:

CHAIN	loads another program and passes variables to the current program.
END	ends a program.
FOR/NEXT	establishes a program loop.
GOSUB	transfers program control to the subroutine.
GOTO	transfers program control to the specified line number.
IF . . . THEN . . . ELSE	evaluates an expression and performs an operation if conditions are met.
ON . . . GOSUB	evaluates an expression and branches to a subroutine.

ON . . . GOTO	evaluates an expression and branches to another program line.
RETURN	returns from a subroutine to the calling program.
STOP	stops program execution.
WHILE . . . WEND	executes statements in a loop as long as a given condition is true.
WAIT	suspends program execution while monitoring the status of a machine input port.

Statements for storing and accessing data on disk:

CLOSE	closes access to a disk file.
FIELD	organizes a direct-access buffer.
GET	gets a record from a direct-access file.
INPUT#	inputs data from a disk file.
LINE INPUT#	inputs an entire line from a disk file.
LSET	moves data (and left-justifies it) to a field in a direct-access file buffer.
OPEN	opens a disk file.
PRINT#	writes data to a sequential disk file.
PRINT# USING	writes data to a disk file using the specified format.
PUT	puts a record into a direct-access file.
RSET	moves data (and right-justifies it) to a field in a direct-access file buffer.
WRITE#	writes data to a sequential file.

Statements for debugging a program:

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

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

CLS	clears the display.
INPUT	inputs data from the keyboard.
LINE INPUT	inputs an entire line from the keyboard.
LIST	lists a program to the display.
LLIST	lists program to line printer.
LPRINT	prints data at the line printer.
PRINT	prints data to the display.
WRITE	prints data on the display.

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

AUTO	automatically numbers program lines.
CALL	calls an assembly-language subroutine.
DELETE	erases program lines from memory.
DEF USR	specifies the starting address of an assembly-language subroutine.
EDIT	edits program lines.
KILL	deletes a disk file.
LOAD	loads a program from disk.
MERGE	merges a disk program with a resident program.
NAME	renames a disk file.
NEW	erases a program from RAM.
OUT	sends a byte to a machine output port.
POKE	writes a byte into a memory location.
RENUM	renumbers a program.
RUN	executes a program.
SAVE	saves a program on disk.
SOUND	generates a sound
SYSTEM	returns to TRSDOS.

Functions

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

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

Numeric Functions (return a number):

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

MEM	returns the amount of memory.
PEEK	returns a byte from a memory location.
RND	returns a pseudorandom number.
SGN	returns the sign.
SIN	calculates the sign.
SQR	calculates the square root.
TAN	computes the tangent.
USR	calls an assembly-language subroutine.
VAL	returns the numeric value of a string.
VARPTR	returns an address for a variable or buffer.

String Functions (return a string value):

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

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

INKEY$	returns the keyboard character.
INPUT$	returns a string of characters from the keyboard.
POS	returns the cursor column position on the display.
ROW	returns the row position on the display.
SPC	prints spaces to the display.
CVD	restores data from a direct disk file to double precision.
CVI	restores data from a direct disk file to integer.
CVS	restores data from a direct disk file to single precision.
EOF	checks for end-of-file.
INPUT$	inputs a string of characters from a sequential disk file.
LOC	returns the current disk file record number.
LOF	returns the disk file's end-of-file.
MKI$	converts an integer value to a string for writing it to a direct-access disk file.
MKS$	converts a single-precision number to a string for writing it to a direct-access file.
MKD$	converts a double-precision value to a string for writing it to a direct-access file.

Chapter 7/ Statements And Functions

ABS

ABS(*number*)	**Function**

Computes the absolute value of *number*.

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

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

Example

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

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

Sample Program

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

ASC

ASC(*string*)	**Function**

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

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

Example

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

prints 65, the ASCII code for "A".

Sample Programs

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

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

ASC can also be used to program the special function keys, as in the following program.

```
100 CLS : PRINT "Enter ANY Keyboard Character : ";
110 IN$ = INKEY$ : IF IN$ = "" THEN GOTO 110
120 A = ASC(IN$)
130 IF A = 129 THEN IN$ = CHR$(13) + "F1 KEY" +
    CHR$(13)
140 IF A = 130 THEN IN$ = CHR$(13) + "F2 KEY" +
    CHR$(13)
150 IF A = 131 THEN IN$ = CHR$(13) + "F3 KEY" +
    CHR$(13)
160 PRINT IN$;
170 GOTO 110
180 END
```

ATN

ATN(*number*)	**Function**

Computes the arctangent of *number* in radians.

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

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

Example

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

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

AUTO

AUTO [***line number***][***,increment***]	**Statement**

Automatically generates a *line number* every time you press (ENTER). Immediately following the line number, you can enter your text for that line.

AUTO begins numbering at *line* and displays the next line using *increment*. The default for both values is 10. A period (.) can be substituted for *line*. In this case, BASIC uses the current line number.

IF AUTO generates a line number that has already been used, it displays an asterisk after the number. To save the existing line, press (ENTER) immediately after the asterisk. AUTO then generates the next line number.

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

Examples

```
AUTO
```

generates lines 10, 20, 30, 40.

```
AUTO 100, 50
```

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

CALL

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

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

Variable contains the address where the subroutine starts in memory. *Variable* may not be an array variable.

Parameter list contains the values that are passed to the external subroutine. *Parameter list* may contain only variables.

A CALL statement with no parameters generates a simple Z-80 "CALL" instruction. The corresponding subroutine should return with a simple "RET".

The method for passing parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. HL contains the address pointing to parameter 1. DE contains the address pointing to parameter 2. BC contains the address pointing to parameter 3.

2. If the number of parameters is greater than 3, they are passed as follows:

   HL contains the address pointing to parameter 1.

   DE contains the address pointing to parameter 2.

   BC points to the low byte of a contiguous data block containing parameters 3 through n (that is, to the low byte of parameter 3).

Note that with this scheme, the subroutine must know how many parameters to expect in order to find them. The calling program is responsible for passing the correct number of parameters.

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

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

See also USR and VARPTR.

Example

```
110 MYROUT = &HD000
120 CALL MYROUT(I,J,K)
```

We assume that D000 is the address for an assembly-language routine. The values of I, J, and K (which we also assume were given elsewhere) are passed to that routine.

CDBL

CDBL(*number*)	**Function**

Converts *number* to double precision.

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

Sample Program

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

CHAIN

Statement
CHAIN [**MERGE**] ***"filespec"*** [,[***line***] [,**ALL**] [,**DELETE** ***line-line***]]

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

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

Line is the first line to be run in the CHAINed program. If omitted, execution begins at the first program line of the CHAINed program.

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

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

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

Examples

```
CHAIN "PROG2"
```

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

```
CHAIN "SUBPROG/BAS", ALL
```

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

Sample Program 1

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING
   USING COMMON TO PASS VARIABLES.
20 REM SAVE THIS MODULE ON DISK AS "PROG1"
   USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1)="VARIABLES IN COMMON MUST BE ASSIGNED "
```

```
 60 A$(2)="VALUES BEFORE CHAINING"
 70 B$(1)="":B$(2)=""
 80 CHAIN "PROG2"
 90 PRINT : PRINT B$(1): PRINT : PRINT B$(2):
    PRINT
100 END
```

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

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

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

```
VARIABLES IN COMMON MUST BE ASSIGNED VALUES
BEFORE CHAINING. NOTE HOW THE OPTION OF
SPECIFYING A STARTING LINE NUMBER WHEN
CHAINING AVOIDS THE DIMENSION STATEMENT IN
'PROG1'.
```

Type NEW and this program:

Sample Program 2

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

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

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

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

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

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

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

NOTE

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

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

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

CHR$

CHR$(*code*)	**Function**

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

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

Examples

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

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

```
PRINT CHR$(16)
```

puts the display into its black-on-white mode, also called reverse video mode. PRINT CHR$(28) returns it to white-on-black and converts all reverse video characters into graphics characters. See Appendix C for more information.

Sample Program

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

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

For a complete list and discussion of output to the video display, see the Character Codes table in Appendix C. See also the sample program given for the ASC function of BASIC.

CINT

CINT(*number*)	**Function**

Converts *number* to integer representation.

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

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

Sample Program

```
PRINT CINT(17.65)
 18
Ready
```

CLEAR

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

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

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

Stack space must also be an integer. This sets aside memory for temporarily storing internal data and addresses during subroutine calls and during FOR/NEXT loops. The default is 512 bytes or one-eighth of the memory available, whichever is smaller. An "Out of memory" error occurs if there is insufficient stack space for program execution.

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

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

Examples:

```
CLEAR
```

clears all variables and closes all files.

```
CLEAR, 45000
```

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

```
CLEAR, 61000, 200
```

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

CLOSE

CLOSE [*buffer*, . . .]	**Statement**

Closes access to a file.

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

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

```
CLOSE buffer
```

has no effect.

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

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

Examples

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

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

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

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

CLS

CLS **Statement**

Clears the screen and moves the cursor to the upper-left corner. All characters on the screen are erased.

Sample Program

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

COMMON

COMMON ***variable, . . .*** **Statement**

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

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

The same variable cannot appear in more than one COMMON statement. To specify array variables, append "( )" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

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

Example

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

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

See also CHAIN.

CONT

CONT	**Statement**

Resumes program execution.

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

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

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

Example

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

Run this program. (To enter the ^, press (CLEAR) (;).) You will be prompted with:

```
?
```

Type:

```
1, 2, 3 (ENTER)
```

The computer displays:

```
Break in 40
```

You can now type any immediate command.

For example:

```
PRINT L
```

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

For example:

```
C = 4
```

changes the value of C in the program. Type:

```
CONT
```

your screen displays: 144.

See also STOP.

COS

COS(*number*)	**Function**

Computes the cosine of *number*.

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

The result is always single precision.

Examples

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

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

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

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

CSNG

CSNG(*number*) — **Function**

Converts *number* to single precision.

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

Example

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

prints .145389

Sample Program

```
280 V# = 876.2345678#
290 PRINT V#; CSNG(V#)
RUN
 876.2345678000001     876.235
Ready
```

CVD, CVI, CVS

Function

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

Convert string values to numeric values.

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

CVD converts an *eight-byte string* to a double-precision number. CVS converts a *four-byte string* to a single-precision number. CVI converts a *two-byte string* to an integer.

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

Examples

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

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

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

Sample Program

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

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

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

DATA

DATA *constant, . . .* — **Statement**

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

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

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

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

Examples

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

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

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

stores four numeric data items.

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

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

Sample Program

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

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

DATE$

DATE$	Function

Returns today's date.

The operator sets the date when TRSDOS is started up.
(This system supports dates between January 1, 1980 and December 31, 1987).

During a program, if you request the date, BASIC displays it in this fashion:

```
03/12/83
```

Sample Program

```
1090 PRINT "Inventory Check:"
1100 IF DATE$ = "01/31/80" THEN PRINT "Today is
     the last day of January 1980. Time to
     perform monthly inventory.": END
```

DEFDBL/INT/SNG/STR

Statement

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

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

NOTE: A type declaration character always takes precedence over a DEF statement.

Examples

```
10 DEFDBL L-P
```

classifies all variables beginning with the letters L through P as double-precision variables. Their values include 17 digits of precision, though only 16 are printed out.

```
10 DEFSTR A
```

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

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

classifies all variables beginning with the letters I through N, W and Z as integer variables. Their values are in the range −32768 to 32767.

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

classifies all variables beginning with the letters I or Q through T as single-precision variables. Their values include seven digits of precision, though only six are printed out.

DEF FN

Statement

DEF FN *function name* [(*variable*, . . .)] =*function definition*

Defines *function name* according to your *function definition*.

Function name must be a valid variable name. The type of variable used determines the type of value the function will return. For example, if you use a single-precision variable, the function will always return single-precision values.

Variable represents those variables in *function definition* that are to be replaced when the function is called. If you enter several variables, separate them by commas.

Function definition is an expression that performs the operation of the function. A variable used in a function definition may or may not appear as *variable*. If it does, BASIC uses its value to perform the function. Otherwise, it uses the current value of the variable.

Once you define and name a function (by using this statement), you can call it and BASIC performs the associated operations.

Examples

```
DEF FNR = RND(90)+9
```

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

```
210 DEF FNW# (A#,B#)=(A#-B#)*(A#-B#)
280 T = FNW#(I#,J#)
```

defines function FNW# in line 210. Line 280 calls that function and replaces parameters A# and B# with parameters I# and J#. (We assume that I# and J# were assigned values elsewhere in the program.)

NOTE: Using a variable as a parameter in a DEF FN statement has no effect on the value of that variable. You may use that variable in another part of the program without interference from DEF FN.

DEF USR

Statement

DEF USR[*digit*]=***address***

Defines the starting address for the assembly-language subroutine identified by *digit*.

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

If you omit *digit*, BASIC assumes USR0.

See also USR, VARPTR and CALL.

Examples

```
DEF USR3 = &H7D00
```

assigns the starting address 7D00 hexadecimal, 32000 decimal, to the USR3 call. When your program calls USR3, control branches to your subroutine beginning at 7D00.

```
DEF USR = (BASE + 16)
```

assigns the starting address of BASE + 16 to the USR0 subroutine.

DELETE

DELETE *line1* - *line2*	**Statement**

Deletes from *line1* through *line2* of a program in memory.

A period (".") can be substituted for either *line1* or *line2* to indicate the current line number.

Examples

```
DELETE 70
```

deletes line 70 from memory. If there is no line 70, an error will occur.

```
DELETE 50-110
```

deletes lines 50 through 110 inclusive.

```
DELETE -40
```

deletes all program lines up to and including line 40.

```
DELETE -.
```

deletes all program lines up to and including the line that has just been entered or edited.

```
DELETE .
```

deletes the program line that has just been entered or edited.

DIM

DIM *array* (*dimension(s)*), *array* (*dimension(s)*), . . .	**Statement**

Sets aside storage for *arrays* with the *dimensions* you specify.

Arrays may be of any type: string, integer, single precision or double precision, depending on the type of variable used to name the array. If no type is specified, the array is classified as single precision.

When you create the array, BASIC reserves space in memory for each element of the array. All elements in a newly- created array are set to zero (numeric arrays) or the null string (string arrays).

NOTE: The lowest element in a dimension is always zero, unless OPTION BASE 1 has been used.

Arrays can be created implicitly, without explicit DIM statements. Simply refer to the desired array in a BASIC statement. For example,

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

creates array A and assigns element A(5) the value of 300. Each dimension of an implicitly-defined array is 11 elements deep, subscripts 0 – 10.

Examples

```
DIM AR(100)
```

sets up a one-dimensional array AR( ), containing 101 elements: AR(0), AR(1), AR(2), . . . , AR(98), AR(99), and AR(100).

NOTE: The array AR( ) is completely independent of the variables AR.

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

sets up a two-dimensional array L1%(,), containing 9 × 26 integer elements, L1%(0,0), L1%(1,0), L1%(2,0), . . . ,L1%(8,0), L1%(0,1), L1%(1,1), . . . ,L1%(8,1), . . . ,L1%(0,25), L1%(1,25), . . . , L1%(8,25).

Two-dimensional arrays like AR(,) can be thought of as a table in which the first subscript specifies a row position, and the second subscript specifies a column position:

```
0,0    0,1    0,2    0,3    ...    0,23    0,24    0,25
1,0    1,1    1,2    1,3    ...    1,23    1,24    1,25
.
.
.
7,0    7,1    7,2    7,3    ...    7,23    7,24    7,25
8,0    8,1    8,2    8,3    ...    8,23    8,24    8,25
```

```
DIM B1(2,5,8), CR(2,5,8), LY$(50,2)
```

sets up three arrays:

B1(,,) and CR (, ,) are three-dimensional, each containing 3*6*9 elements.

LY(,) is two-dimensional, containing 51*3 string elements.

EDIT

EDIT *line*	**Statement**

Enters the edit mode so that you can edit *line*.

See the chapter on the "Edit Mode" for more information.

Examples

```
EDIT  100
```

enters edit mode at line 100.

```
EDIT .
```

enters edit mode at current line.

END

END	Statement

Ends execution of a program.

This statement may be placed anywhere in the program. It forces execution to end at some point other than the last sequential line.

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

Sample Program

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

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

EOF

EOF(*buffer*)	Function

Detects the end of a file.

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

EOF(*buffer*) returns 0 (false) when the EOF record has not been read yet, and −1 (true) when it has been read. The buffer number must access an open file.

Sample Program

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop.

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

ERASE

ERASE *array*, . . .	Statement

Erases one or more *arrays* from a program.

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

If one of the parameters of ERASE is a variable name which is not used in the program, an "Illegal Function Call" occurs.

Example

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

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

ERL

ERL	**Statement**

Returns the line in which an error has occurred.

This function is primarily used inside an error-handling routine. If no error has occurred when ERL is called, line number 0 is returned. Otherwise, ERL returns the line number in which the error occurred. If the error occurred in the command mode, 65535 (the largest number representable in two bytes) is returned.

Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number for future use.

For an example of how to use ERL in a program, see ERROR.

ERR

ERR	Statement

Returns the error code (if an error has occurred).

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO. See Appendix D for a list of Error Codes.

Example

```
IF ERR = 7 THEN 1000 ELSE 2000
```

branches the program to line 1000 if the error is an "Out of Memory" error (code 7); if it is any other error, control goes instead to line 2000.

For an example of how to use ERR in a program, see ERROR.

ERRS$

ERRS$	Function

Returns a system error number and message.

This function returns the number and description of the TRSDOS error that caused the latest BASIC disk-related error. If no TRSDOS error has occurred, ERRS$ returns a null string.

Example

```
PRINT  "THE LATEST TRSDOS ERROR IS  "; ERRS$
```

prints the latest error number message.

ERROR

ERROR ***code***	**Statement**

Simulates a specified error during program execution.

Code is an integer expression in the range 0 to 255 specifying one of BASIC's error codes.

This statement is mainly used for testing an ON ERROR GOTO routine. When the computer encounters an ERROR code statement, it proceeds as if the error corresponding to that code had occurred. (Refer to Appendix D for a listing of Error Codes and their meanings).

Example

```
ERROR 1
```

a "Next Without For" error (code 1) "occurs" when BASIC reaches this line.

Sample Program

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET"; B
130 IF B>5000 THEN ERROR 21 ELSE GOTO 420
400 IF ERR = 21 THEN PRINT "HOUSE LIMIT IS
    $5000"
410 IF ERL = 130 THEN RESUME 500
420 S = S+B
430 GOTO 120
500 PRINT "THE TOTAL AMOUNT OF YOUR BET IS";S
510 END
```

This program receives and totals bets until one of them exceeds the house limit.

EXP

EXP(*number*)	**Function**

Calculates the natural exponent of *number*.

Returns e (base of natural logarithms) to the power of *number*. This is the inverse of the LOG function; therefore, *number* = EXP(LOG(*number*)). The *number* you supply must be less than or equal to 87.3365.

The result is always single precision.

Example

```
PRINT EXP(-2)
```

prints the exponential value .135335.

Sample Program

```
310 INPUT "NUMBER"; N
320 PRINT "E RAISED TO THE N POWER IS" EXP(N)
```

FIELD

FIELD *buffer, length* AS *field name, . . .*	Statement

Divides a direct-access *buffer* into one or more fields. Each field is identified by *field name* and is the *length* you specify.

Field name must be a string variable.

This divides a direct file buffer so that you can send data from memory to disk and disk to memory. FIELD must be run prior to GET or PUT.

Before "fielding" a buffer, use an OPEN statement to assign that buffer to a particular disk file. (The direct access mode, i.e., OPEN "D", . . . must be used.) The sum of all field lengths should equal the record length assigned when the file was OPENed.

You may use the FIELD statement any number of times to "re-field" a file buffer. "Fielding" a buffer does not clear the buffer's contents; only the means of accessing it. Also, two or more field names can reference the same area of the buffer.

See also the chapter on "Disk Files", OPEN, CLOSE, PUT, GET, LSET, and RSET.

Example

```
FIELD 3, 128  AS A$, 128 AS B$
```

tells BASIC to assign two 128-byte fields to the variables A$ and B$. If you now print A$ or B$, you will see the contents of the field. Of course, this value would be meaningless unless you have previously used GET to read a 256-byte record from disk.

NOTE: All data — both strings and numbers — must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI, MKS$/CVS, and MKD$/CVD) for converting numbers to strings and strings to numbers.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS
ST$, 7 AS ZP$
```

assigns the first 16 bytes of buffer 3 to field NM$; the next 25 bytes to AD$; the next 10 to CY$; the next 2 to ST$; and the next 7 to ZP$.

FIX

FIX(*number*)	**Function**

Returns the truncated integer of *number*.

All digits to the right of the decimal point are simply chopped off, so the resultant value is a whole number. For a negative, non-whole number X, FIX(X) = INT(X) + 1. For all others, FIX(X) = INT(X).

The result is the same precision as the argument (except for the fractional portion).

Examples

```
PRINT FIX (2.6)
```

prints 2.

```
PRINT FIX(-2.6)
```

prints -2.

FOR/NEXT

Statement
FOR *variable* = *initial value* TO *final value* [STEP *increment*]
NEXT [*variable*]

Establishes a program loop.

A loop allows for a series of program statements to be executed over and over a specified number of times.

BASIC executes the program lines following the FOR statement until it encounters a NEXT. At this point, it increases *variable* by STEP *increment*. If the value of *variable* is less than or equal to *final value*, BASIC branches back to the line after FOR, and repeats the process. If *variable* is greater than *final value*, it completes the loop and continues with the statement after NEXT.

If *increment* has a negative value, then the final value of *variable* is actually lower than the initial value. BASIC always sets the final value for the loop variable before setting the initial value.

NOTE: BASIC skips the body of the loop if *initial value* times the sign of STEP *increment* exceeds *final value* times the sign of STEP *increment*.

Example

```
20 FOR H=1 TO -10 STEP -2
30 PRINT H
40 NEXT H
```

the initial value of H times the sign of STEP increment is greater than the final value of H times the sign of STEP increment, therefore BASIC skips the body of the loop. (The sign of STEP increment is negative in this case.)

Sample Program

```
820 I=5
830 FOR I = 1 TO I + 5
840 PRINT I;
850 NEXT
RUN
```

this loop is executed ten times. It produces the following output:

```
1  2  3  4  5  6  7  8  9  10
```

Nested Loops

FOR/NEXT loops may be "nested". That is, a FOR . . . NEXT loop may be placed within the context of another FOR . . . NEXT loop.

The NEXT statement for the inside loop must appear before the NEXT for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

Sample Program

```
880 FOR I = 1 TO 3
890 PRINT "OUTER LOOP"
900 FOR J = 1 TO 2
910 PRINT "INNER LOOP"
920 NEXT J
930 NEXT I
```

This program performs three "outer loops" and within each, two "inner loops".

The NEXT statement can be used to close nested loops by listing the counter variables (but make sure not to type the variables out of order). For example, delete line 920 and change 930 to:

```
NEXT J, I
```

NOTE: In nested loops, if the variable(s) in the NEXT statement is omitted, the NEXT statement matches the most recent FOR statement.

FRE

FRE(*dummy number*) or (*dummy string*)	**Function**

Returns the number of bytes in memory not being used by BASIC.

NOTE: FRE forces a "garbage collection" before returning the number of free bytes. This may take up to one and a half minutes. Using FRE periodically results in shorter delays for each garbage collection.

Examples

```
PRINT FRE("44")
```

prints the amount of memory left.

```
PRINT FRE(44)
```

prints the amount of memory left.

GET

GET *buffer* [*,record*]	**Statement**

Gets a *record* from a direct-access disk file and places it in a *buffer*.

Before using GET, you must OPEN the file and assign it a buffer.

When BASIC encounters GET, it reads the record number from the file and places it into the buffer. The actual number of bytes read equals the record length set when the file is OPENed.

If *record* is omitted, BASIC gets the next record (after the last GET) and reads it into the buffer.

Examples

```
GET 1
```

gets the next record into buffer 1.

```
GET 1, 25
```

gets record 25 into buffer 1.

GOSUB

GOSUB *line*	**Statement**

Goes to a subroutine, beginning at *line*.

You can call subroutine as many times as you want. When the computer encounters RETURN in the subroutine, it returns control to the statement which follows GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement. Every subroutine must end with a RETURN.

Example

```
GOSUB 1000
```

branches control to the subroutine at 1000.

Sample Program

```
260 GOSUB 280
270 PRINT "BACK FROM SUBROUTINE": END
280 PRINT "EXECUTING THE SUBROUTINE"
290 RETURN
```

transfers control from line 260 to the subroutine beginning at line 280. Line 290 instructs the computer to return to the statement immediately following GOSUB.

GOTO

GOTO *line*	**Statement**

Goes to the specified *line*.

When used alone, GOTO *line* results in an unconditional (automatic) branch. However, test statements may precede the GOTO to effect a conditional branch.

You can use GOTO in the command mode as an alternative to RUN. This lets you pass values assigned in the command mode to variables in the execute mode.

Example

```
GOTO 100
```

transfers control automatically to line 100.

Sample Program

```
10 READ R
20 IF R = 13 THEN END
30 PRINT "R=";R
40 A=3.14*R^2
50 PRINT "AREA =";A
60 GOTO 10
70 DATA 5,7,12, 13
RUN
```

line 10 reads each of the data items in line 60; line 50 returns program control to line 10. This enables BASIC to calculate the area for each of the data items, until it reaches item 13.

NOTE: To enter the ^ symbol, press (CLEAR) (;).

HEX$

HEX$(*number*) **Function**

Calculates the hexadecimal value of *number*.

HEX$ returns a string which represents the hexadecimal value of the argument. The value returned is like any other string: it cannot be used in a numeric expression.That is, you cannot add hex strings. You can concatenate them, though.

Examples

```
PRINT HEX$(30), HEX$(50), HEX$(90)
```

prints the following strings:

```
1E          32          5A
```

```
Y$ = HEX$(X/16)
```

Y$ is the hexadecimal string representing the integer quotient X/16.

IF . . . THEN . . . ELSE

IF *expression* THEN *statement(s)* or *line*
[ELSE *statement(s)* or *line*] **Statement**

Tests a conditional expression and makes a decision regarding program flow.

If *expression* is true, control proceeds to the THEN *statement* or *line*. If not, control jumps to the matching ELSE *statement, line*, or down to the next program line.

Examples

```
IF X > 127 THEN PRINT "OUT OF RANGE" : END
```

passes control to PRINT, then to END if X is greater than 127. If X is not greater than 127, control jumps down to the next line in the program, skipping the PRINT and END statements.

```
If A < B THEN PRINT "A < B" ELSE PRINT "B < A"
```

tests the first expression, if true, prints "A < B". Otherwise, the program jumps to the ELSE statement and prints "B < A".

```
IF X > 0 AND Y <> 0 THEN Y = X + 180
```

assigns the value X + 180 to Y if both expressions are true. Otherwise, control passes directly to the next program line, skipping the THEN clause.

```
IF A$ = "YES" THEN 210 ELSE IF A$ = "NO" THEN
    400 ELSE 370
```

branches to line 210 if A$ is YES. If not, the program skips over to the first ELSE, which introduces a new test. If A$ is NO, then the program branches to line 400. If A$ is any value besides NO or YES, the program branches to line 370.

Sample Program

IF THEN ELSE statements may be nested. However, you must take care to match up the IFs and ELSEs. (If the statement does not contain the same number of ELSE's and IF's, each ELSE is matched with the closest unmatched IF.)

```
1040 INPUT "ENTER TWO NUMBERS"; A, B
1050 IF A <= B THEN IF A < B THEN PRINT A;
     ELSE PRINT "NEITHER"; ELSE PRINT B;
1060 PRINT "IS SMALLER THAN THE OTHER"
```

This program prints the relationship between the two numbers entered.

INKEY$

INKEY$	**Function**

Returns a keyboard character.

Returns a one-character string from the keyboard without having to press (ENTER). If no key is pressed, a null string (length zero) is returned. Characters typed to INKEY$ are not echoed to the display.

INKEY$ is invariably put inside some sort of loop. Otherwise a program execution would pass through the line containing INKEY$ before a key could be pressed.

Example

```
10 A$ = INKEY$
20 IF A$ = "" THEN 10
```

This causes the program to wait for a key to be pressed.

INP

INP(*port*)	**Function**

Returns the byte read from a *port*.

INP is the complementary function of the OUT statement.

Port may be any integer from 0 to 255. For information on assigned ports, see the Technical Reference Manual.

Example

```
100 A=INP(42)
```

INPUT

INPUT [*prompt string*;] variable1, variable2, . . .	**Statement**

Inputs data from the keyboard into one or more *variables*.

When BASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type data.

INPUT may specify a list of string or numeric variables, indicating string or numeric data items to be input. For instance, INPUT X$, X1, Z$, Z1 calls for you to input a string literal, a number, another string literal, and another number, in that order.

The number of data items you supply must be the same as the number of variables specified. You must separate data items by commas.

Responding to INPUT with too many items, or with the wrong type of value (including numeric type), causes BASIC to print the message "?Redo from start". No values are assigned until you provide an acceptable response.

If a *prompt string* is included, BASIC prints it, followed by a question mark. This helps the person inputting the data to enter it correctly. If instead of a semicolon, you type a comma after *prompt string*, BASIC suppresses the question mark when printing the prompt. *Prompt string* must be enclosed in quotes. It must be typed immediately after INPUT.

Examples

```
INPUT Y%
```

when BASIC reaches this line, you must type any number and press (ENTER) before the program will continue.

```
INPUT SENTENCE$
```

when BASIC reaches this line, you must type in a string. The string wouldn't have to be enclosed in quotation marks unless it contained a comma, a colon, or a leading blank.

```
INPUT "ENTER YOUR NAME AND AGE (NAME, AGE)";
N$, A
```

would print a message on the screen which would help the person at the keyboard to enter the right kind of data.

Sample Program

```
50 INPUT "HOW MUCH DO YOU WEIGH"; X
60 PRINT "ON MARS YOU WOULD WEIGH ABOUT"
   CINT(X * .38) "POUNDS."
```

INPUT#

Statement

INPUT# ***buffer, variable, . . .***

Inputs data from a sequential disk file and stores it in a program *variable*.

Buffer is the number used when the file was OPENed for input.

Variable contains the variable name(s) that will be assigned to the item(s) in the file.

With INPUT#, data is input sequentially. That is, when the file is OPENed, a pointer is set to the beginning of the file. The pointer advances each time data is input. To start reading from the beginning of the file again, you must close the file buffer and re-OPEN it.

INPUT# doesn't care how the data was placed on the disk — whether a single PRINT# statement put it there, or whether it required ten different PRINT# statements. What matters to INPUT# is the position of the terminating characters and the EOF marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Numeric values: BASIC begins input at the first character which is neither a space nor a carriage return. It ends input when it encounters a space, carriage return, or a comma.

String values: BASIC begins input with the first character which is neither a space nor carriage return. It ends input when it encounters a carriage return or comma. One exception to this rule: If the first character is a quotation mark("), the string will consist of all characters between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character.

If the end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Examples

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

sequentially inputs two numeric data items from disk and places them in A and B. Buffer #1 is used.

```
INPUT#4, A$, B$, C$
```

sequentially inputs three string data items from disk and places them in A$, B$, and C$. Buffer #4 is used.

INPUT$

Statement

INPUT$(*number* [,*buffer*])

Inputs a string of characters from either the keyboard or a sequential disk file.

Number is the number of characters to be input. It must be a value in the range 1 to 255. *Buffer* is a buffer which accesses a sequential input file.

INPUT$(*number*) inputs a string of characters from the keyboard. When the program reaches this line, it stops until you (or any operator) type *number* characters. (You don't need to press (ENTER) to signify end-of-line.) The character(s) you type are not displayed on the screen. Any character, except (BREAK), is accepted for input. No characters are echoed.

INPUT$(*number, buffer*) inputs a string from a sequential disk file. *Buffer* is the buffer associated with that disk file.

Examples

```
A$ = INPUT$(5)
```

assigns a string of five keyboard characters to A$. Program execution is halted until the operator types five characters.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A$. The characters are read from the disk file associated with buffer 3.

Sample Programs

This program shows how you could use INPUT$ to have an operator input a password for accessing a protected file. By using INPUT$, the operator can type in the password without anyone seeing it on the video display. (To see the full file specification, run the program, then type PRINT F$.)

```
110 LINE INPUT "TYPE IN THE FILESPEC/EXT"; F$
120 PRINT "TYPE IN THE PASSWORD -- MUST TYPE 8
    CHARACTERS: ";
130 P$ = INPUT$(8)
140 F$ = F$ + "." + P$
```

In the program below, line 100 OPENs a sequential input file (which we assume has been previously created). Line 200 retrieves a string of 70 characters from the file and stores them in T$. Line 300 CLOSEs the file.

```
100 OPEN "I", 2, "TEST/DAT"
200 T$ = INPUT$(70,2)
300 CLOSE
```

INSTR

INSTR([*integer*,] ***string1*, *string2*)** **Function**

Searches for the first occurrence of *string2* in *string1*, and returns the position at which the match is found.

Integer specifies a position in *string1*. If used it must be a value in the range 1 to 255.

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise, it returns zero. Note that the entire substring must be contained in the search string, or zero is returned.

Optional *integer* sets the position for starting the search. If omitted, INSTR starts searching at the first character in *string1*.

Examples

In these examples, A$ = "LINCOLN":

```
INSTR(A$, "INC")
```

returns a value of 2.

```
INSTR(A$, "12")
```

returns a zero.

```
INSTR(A$, "LINCOLNABRAHAM")
```

returns a zero. For a slightly different use of INSTR, look at:

```
INSTR (3, "1232123", "12")
```

which returns 5.

Sample Program

The program below uses INSTR to search through the addresses contained in the program's DATA lines. It counts the number of addresses with a specified county zip code (761—) and returns that

number. The zip code is preceded by an asterisk to distinguish it from the other numeric data found in the address.

```
360 RESTORE
370 COUNTER = 0
390 READ ADDRESS$
395 IF ADDRESS$ = "$END" THEN 410
400 IF INSTR(ADDRESS$, "*761") <> 0 THEN COUNTER =
    COUNTER + 1 ELSE 390
405 GOTO 390
410 PRINT "NUMBER OF TARRANT COUNTY, TX
    ADDRESSES IS" COUNTER: END
420 DATA "5950 GORHAM DRIVE, BURLESON, TX
    *76148"
430 DATA "71 FIRSTFIELD ROAD, GAITHERSBURG, MD
    *20760"
440 DATA "1000 TWO TANDY CENTER, FORT WORTH,
    TX *76102"
450 DATA "16633 SOUTH CENTRAL EXPRESSWAY,
    RICHARDSON, TX *75080"
460 DATA "$END"
```

INT

INT(*number*)	**Function**

Converts *number* to integer value.

This function returns the largest integer which is not greater than the *number. Number* may be an expression.

The result has the same precision as the argument except for the fractional portion. *Number* is not limited to the range –32768 to 32767.

Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT (-12.11)
```

prints −13.

KILL

KILL ***"filespec"***	**Statement**

"Kills" (deletes) *filespec* from disk.

You may KILL any type of disk file. However, if the file is currently OPEN, a "File already open" error occurs. You must CLOSE the file before deleting it.

Example

```
KILL "FILE/BAS"
```

deletes this file from the first drive which contains it.

```
KILL "DATA:2"
```

deletes this file from Drive 2 only.

LEFT$

LEFT$(***string,integer*****)**	**Function**

Returns the leftmost *integer* characters of *string*.

If *integer* is equal to or greater than LEN (string), the entire string is returned.

Examples:

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

since BIG FIERCE DOG is less than 20 characters long, the whole phrase is printed.

Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When this is run, BASIC prints:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

Line 750 gets the three leftmost characters of A$ and stores them in B$. Line 760 prints these three characters, a string, and the original contents of A$.

LEN

LEN(*string*)	**Function**

Returns the number of characters in *string*.

Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

LET

[**LET**] ***variable* = *expression***

Statement

Assigns the value of *expression* to *variable.*

BASIC doesn't require assignment statements to begin with LET, but you might want to use LET to be compatible with versions of BASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X  = X - Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P = "P
```

LINE INPUT

LINE INPUT[*prompt string*;] ***string variable***	**Statement**

Inputs an entire line (up to 254 characters) from the keyboard.

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters (commas, quotation marks, etc.).

LINE INPUT (the space is *not* optional) is similar to INPUT, except:

— The computer does not display a question mark when waiting for input.
— Each LINE INPUT statement can assign a value to only one variable.
— Commas and quotes can be used as part of the string input.
— Leading blanks are not ignored — they become part of variable.

The only way to terminate the string input is to press (ENTER).

Some situations require that you input commas, quotes, and leading blanks as part of the data. LINE INPUT serves well in such cases.

Examples:

```
LINE INPUT A$
```

inputs A$ without displaying any prompt.

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

displays a prompt message and inputs data. Commas do not terminate the input string, as they would in an INPUT statement.

You may abort a LINE INPUT statement by pressing (BREAK). BASIC returns to command level and displays Ready. Typing CONT resumes execution at LINE INPUT.

LINE INPUT#

Statement

LINE INPUT# ***buffer, variable***

Inputs an entire line of data from a sequential disk file to a string *variable*.

Buffer is the number under which the file was OPENed.

This statement is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

— the end-of-file
— the 255th data character

Other characters encountered — quotes, commas, leading blanks — are included in the string.

Example

If the data on disk looks like this:

```
10 CLEAR 500
20 OPEN "I", 1, "PROG"
```

then the statement

```
LINE INPUT#1, A$
```

could be used repetitively to read each program line, one at a time.

LIST

LIST [*startline*]-[*endline*]	**Statement**

Lists a program in memory to the display.

Startline specifies the first line to be listed. If omitted, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If omitted, BASIC ends with the last line in your program.

You can substitute period (.) for either *startline* or *endline* to signify current line number.

Examples

```
LIST
```

displays the entire program. To stop the automatic scrolling, press SHIFT @. This freezes the display. Press any key to continue the listing.

```
LIST 50
```

displays line 50.

```
LIST 50-85
```

displays lines in the range 50-85.

```
LIST .-
```

displays the program line that has just been entered or edited, and all higher-numbered lines.

```
LIST -227
```

displays all lines up to and including 227.

LLIST

LLIST [*startline*]-[*endline*]	**Statement**

Lists program lines in memory to the printer.

The only difference between LLIST and LIST is that LLIST lists the lines on printer. See LIST.

Examples

```
LLIST
```

lists the entire program to the printer. To stop this process, press (SHIFT)(@). This causes a temporary halt in the computer's output to the printer. Press any key to continue printing.

```
LLIST 68-90
```

prints lines in the range 68-90.

LOAD

LOAD *"filespec"* [,R]	**Statement**

Loads *filespec*, a BASIC program, into memory.

The R option tells BASIC to run the program. (LOAD with the R option is equivalent to the command RUN *filespec*, R.)

LOAD without the R option wipes out any resident BASIC program, clears all variables, and CLOSES all OPEN files. LOAD with the R option leaves all OPEN files open and runs the program automatically.

You can use either of these commands inside programs to allow program chaining (one program calling another).

If you attempt to LOAD a non-BASIC file, a "Direct statement in file" error will occur.

Example

```
LOAD  "PROG1/BAS:2"
```

loads PROG1/BAS from Drive 2. BASIC then returns to the command mode.

```
LOAD  "PROG1/BAS"
```

loads PROG1/BAS. Since no drive is specified, BASIC begins searching for it in Drive 0.

LOC

LOC(*buffer*)	**Function**

Returns the current record number.

Buffer is the buffer under which the file was OPENed.

LOC is used to determine the current record number, that is, the number of the last record processed since the file was OPENed. It returns the record number accessed by the last GET or PUT statement.

LOC is also valid for sequential files. It returns the number of sectors (256-byte block) read from or written to the file since the file was OPENed.

Example

```
IF LOC(1)>55 THEN END
```

if the current record number is greater than 55, ends program execution.

Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD"
     LOC(1): CLOSE: END
1340 GOTO 1320
```

This is a **portion** of a program. Elsewhere the file has been OPENed and FIELDed. N$ is a field variable. If N$ matches A$, the record number in which it was found is printed.

LOF

LOF(*buffer*)	**Function**

Returns the end-of-file record number.

Buffer is the number under which a file was OPENed.

This function tells you the number of the last record in a direct-access file.

Example

```
Y = LOF(5)
```

assigns the last record number to variable Y.

Sample Programs

During direct access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

```
1540 OPEN "R", 1, "UNKNOWN/TXT", 255
1550 FIELD 1, 255 AS A$
1560 FOR I% = 1 TO LOF(1)       'LOF(1) = HIGHEST
1570 GET 1, I%                  'RECORD NUM. TO BE
1580 PRINT A$                   'ACCESSED
1590 NEXT I%
1600 CLOSE
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an error.

When you want to add to the end of a file, LOF tells you where to start adding:

```
1600 I% = LOF(1) + 1     'HIGHEST EXISTING RECORD
1610 PUT 1, I%           'ADD NEXT RECORD
```

LOG

LOG(*number*)	**Function**

Computes the natural logarithm of *number*.

This is the inverse of the EXP function. The result is always in single precision.

Examples

```
PRINT LOG(3.14159)
```

prints the value 1.14473.

```
Z = 10 * LOG(Ps/P1)
```

performs the indicated calculation and assigns the value to Z.

Sample Program

This program demonstrates the use of LOG. It utilizes a formula taken from space communications research.

```
540 INPUT "DISTANCE SIGNAL MUST TRAVEL
    (MILES)"; D
550 INPUT "SIGNAL FREQUENCY (GIGAHERTZ)"; F
560 L = 96.58 + (20 * LOG(F)) + (20 * LOG(D))
570 PRINT "SIGNAL STRENGTH LOSS IN FREE SPACE
    IS" L "DECIBELS."
```

LPOS

LPOS(*number*)	**Function**

Returns the logical position of the line printer's print head within the line printer's buffer.

Number is a dummy argument.

This function does not necessarily give the physical position of the print head.

Example

```
100 IF LPOS(X)>60 THEN LPRINT
```

LPRINT, LPRINT USING

LPRINT ***data, . . .*** **LPRINT USING** ***format*****;** ***data, . . .***	**Statement**

Prints *data* on the printer.

See PRINT and PRINT USING for more information.

Examples

```
LPRINT (A * 2)/3
```

prints the value of expression (A * 2)/3 on the printer.

```
LPRINT TAB(50) "TABBED 50"
```

moves the line printer carriage to TAB position 50 and prints "TABBED 50". (Refer to the TAB function).

```
LPRINT USING "#####.#"; 2.17
```

sends the formatted value bbbb2.2 to the line printer.

LSET

Statement

LSET *field name* = *data*

Sets *data* in a direct-access buffer *field name*.

Before using LSET, you must have used FIELD to set up buffer fields.

See also the chapter on "Disk Files", OPEN, CLOSE, FIELD, GET, PUT, and RSET.

Example

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements

```
LSET NM$ = "JIM CRICKET, JR."
LSET AD$ = "2000 EAST PECAN ST."
```

set the data in the buffer as follows:

```
JIMbCRICKET,JR.bbb   2000bEASTbPECANbST.bbbbbb
```

Notice that filler blanks were placed to the right of the data strings in both cases. If we had used RSET statements instead of LSET, the filler spaces would have been placed to the left. This is the only difference between LSET and RSET.

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored. This applies to both LSET and RSET.

MEM

MEM	Function

Returns the amount of memory.

MEM performs the same function as FRE. It returns the number of unused and unprotected bytes in memory.

This function may be used in the immediate mode to see how much space a resident program occupies. It may also be used inside a program to avert "Out of memory" errors. MEM requires no argument.

Example

```
PRINT MEM
```

Enter this command in the immediate mode (no line number is needed). The number returned indicates the amount of leftover memory; that is, memory not being used to store programs, variables, strings, the stack, or not reserved for object files.

Sample Program

```
1610 IF MEM < 80 THEN 1630
1620 DIM A(15)
1630 REM       PROGRAM CONTINUES HERE
```

If fewer than 80 bytes of memory are left, control switches to another part of the program. Otherwise, an array of 16 elements is created.

MERGE

MERGE *"filespec"*	**Statement**

Loads *filespec*, a BASIC program, and merges it with the program currently in memory.

Filespec specifies a BASIC file in ASCII format (a program saved with the A option). If *filespec* is a constant, it must be enclosed in quotes.

Program lines in the disk program are inserted into the resident program in sequential order. For example, suppose that three of the lines from the disk program are numbered 75, 85 and 90, and three of the lines from the current program are numbered 70, 80, and 90. When MERGE is used on the two programs, this portion of the new program will be numbered 70, 75, 80, 85, 90.

If line numbers on the disk program coincide with line numbers in the resident program, the disk program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

Example

Suppose you have a BASIC program on disk, PROG2/TXT (saved in ASCII), which you want to merge with the program you've been working on in memory. Then we use:

```
MERGE  "PROG2/TXT"
```

merges the two programs.

Sample Programs

MERGE provides a convenient means of putting program modules together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

Suppose the following program is in memory:

```
80 REM          MAIN PROGRAM
90 REM LINE NUMBER RESERVED FOR SUBROUTINE HOOK
100 REM         PROGRAM LINE
110 REM         PROGRAM LINE
120 REM         PROGRAM LINE
130 END
```

And suppose the following subroutine, SUB/TXT, is stored on disk in ASCII format:

```
90 GOSUB 1000 SUBROUTINE HOOK
1000 REM        BEGINNING OF SUBROUTINE
1010 REM        SUBROUTINE LINE
1020 REM        SUBROUTINE LINE
1030 REM        SUBROUTINE LINE
1040 RETURN
```

You can MERGE the subroutine with the main program with:

```
MERGE "SUB/TXT"
```

and the new program in memory is:

```
80   REM        MAIN PROGRAM
90   GOSUB   1000 SUBROUTINE HOOK
100  REM        PROGRAM LINE
110  REM        PROGRAM LINE
120  REM        PROGRAM LINE
130  END
1000 REM        BEGINNING OF SUBROUTINE
1010 REM        SUBROUTINE LINE
1020 REM        SUBROUTINE LINE
1030 REM        SUBROUTINE LINE
1040 RETURN
```

MID$

Statement

MID$(*oldstring, position* [*,length*]) = *replacement string*

Replaces a portion of an *oldstring* with *replacement string*.

Oldstring is the variable name of the string you want to change.

Position is a number specifying the position of the first character to be changed.

Length is a number specifying the number of characters to be replaced.

Replacement string is the string to replace a portion of *oldstring*.

The length of the resultant string is always the same as the original string. If *replacement string* is shorter than *length*, the entire replacement string is used.

Examples:

```
A$ = "LINCOLN"

MID$ (A$, 3, 4) = "12345": PRINT A$
```

returns LI1234N.

```
MID$ (A$, 5) = "01": PRINT A$
```

returns LINC01N.

```
MID$ (A$, 1, 3) = "***": PRINT A$
```

returns ***COLN.

MID$

MID$(*string, integer* [*,number*]) — **Function**

Returns a substring of *string*, beginning at position *integer*.

If *integer* is greater than the number of characters in *string*, MID$ returns a null string.

Number is the number of characters in the substring. If omitted, BASIC returns all right most characters, beginning with the character at position *integer*.

Examples

If A$ = "WEATHERFORD" then

```
PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER
    (NNN-NNN-NNNN)"; PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first three digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

MKD$, MKI$, MKS$

Function

MKI$(*integer expression*)
MKS$(*single-precision expression*)
MKD$(*double-precision expression*)

Convert numeric values to string values.

Any numeric value that is placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These three functions are the inverse of CVD, CVI, and CVS. The byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable.

MKD$ returns an eight-byte string; MKI$ returns a two-byte string; and MKS$ returns a four-byte string.

Example

```
LSET AVG$ = MKS$(0.123)
```

Sample Program

```
1350 OPEN "D", 1, "TEST/DAT", 14
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKD$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1, 1
1410 CLOSE 1
```

For a program that retrieves the data from TEST/DAT, see CVD/CVI/CVS.

NAME

Statement

NAME *old filespec* AS *new filespec*

Renames *old filespec* as *new filespec*.

With this statement, the data in the file is left unchanged. The *new filespec* may not contain a password or drive specification.

Example

```
NAME "FILE" AS "FILE/OLD"
```

renames FILE as FILE/OLD.

```
NAME B$ AS A$
```

renames B$ as A$.

NEW

Statement

NEW

Deletes the program currently in memory and clears all variables.

NEW displays a new (clear) screen and returns you to the command mode.

Example

```
NEW
```

OCT$

OCT$(*number*) — **Function**

Computes the octal value of *number*.

OCT$ returns a string which represents the octal value of *number*. The value returned is like any other string — it cannot be used in a numeric expression.

Examples

```
PRINT OCT$(30), OCT$(50), OCT$(90)
```

prints the following strings:

```
36      62      132
```

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

ON ERROR GOTO

ON ERROR GOTO *line* — **Statement**

Transfers control to *line* if an error occurs.

This lets your program "recover" from an error and continue execution. (Normally, you have a particular type of error in mind when you use the ON ERROR GOTO statement).

ON ERROR GOTO has no effect unless it is executed before the error occurs. To disable it, execute an ON ERROR GOTO 0. If you use ON ERROR GOTO 0 inside an error-trapping routine, BASIC stops execution and prints an error message.

The error-handling routine must be terminated by a RESUME statement. See RESUME.

Example

```
10 ON ERROR GOTO 1500
```

branches program control to line 1500 if an error occurs anywhere after line 10.

For the use of ON ERROR GOTO in a program, see the sample program for ERROR.

ON . . . GOSUB

ON *expression* GOSUB *line, . . .*	**Statement**

Calls the subroutine at the *line* based on the value of *expression*.

Expression is a numeric expression between 0 and 255, inclusive. For example, if *expression*'s value is three, the third line number in the list is the destination of the branch.

If *expression's* value is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If *expression* is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
ON Y GOSUB 1000, 2000, 3000
```

If Y = 1, the subroutine beginning at 1000 is called. If Y = 2, the subroutine at 2000 is called. If Y = 3, the subroutine at 3000 is called.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

ON . . . GOTO

ON *expression* GOTO *line, . . .*	**Statement**

Goes to the *line* specified by the value of *expression*.

Expression is a numeric expression between 0 and 255.

This statement is very similar to ON . . . GOSUB. However, instead of branching to a subroutine, it branches control to another program line.

The value of *expression* determines to which line the program will branch. For example, if the value is four, the fourth line number in the list is the destination of the branch. If there is no fourth line number, control passes to the next statement in the program.

If the value of expression is negative or greater than 255, an "Illegal function call" error occurs. Any amount of line numbers may be included after GOTO.

Example

```
ON MI GOTO 150, 160, 170, 150, 180
```

tells BASIC to "Evaluate MI;
if the value of MI equals one then go to line 150;
if it equals two, then go to 160;
if it equals three, then go to 170;
if it equals four, then go to 150;
if it equals five, then go to 180;
if the value of MI doesn't equal any of the numbers one through five, advance to the next statement in the program".

OPEN

OPEN *mode, buffer, "filespec"* [*,record length*]	**Statement**

Opens a disk file.

Mode is a string expression whose first character is one of the following:

O for sequential output mode
I for sequential input mode
E for sequential output and extend mode
D or R for direct input/output mode

Buffer is an integer between 1 and 15. It specifies which area in memory you will use to access the file.

Filespec specifies a TRSDOS file.

Record length is an integer which sets the record length for direct-access files. The default is 256 bytes.

Once you have assigned a buffer to a file with the OPEN statement, that buffer cannot be used in another OPEN statement. You must first CLOSE the first file.

Examples

```
OPEN "D", 2, "DATA/BAS.SPECIAL"
```

opens the file DATA/BAS in direct-access mode, with the password SPECIAL. Buffer 2 is used. If DATA/BAS does not exist, it is created on the first non write-protected drive. The record length is 256 bytes.

```
OPEN "D", 5, "TEXT/BAS", 64
```

opens the file TEXT/BAS for direct access. Buffer 5 is used. The record length is 64. If this length does not match the record length assigned to TEXT/BAS when the file was originally OPENed, an error occurs.

```
OPEN "O", 7, "INV/CONT"
```

opens the sequential file "INV/CONT" for output. If "INV/CONT" does not exist, it is created. Information is written to the file sequentially, starting at the first byte. If the file does exist, any new information is written over the existing information; the file's previous contents are lost.

```
OPEN "E", 1, "LIST/EMP"
```

opens the file LIST/EMP and extends it by appending new data to the end of the file. If "LIST/EMP" does not exist, OPEN "E" works the same way as OPEN "O".

```
OPEN "I", 8, "MGT"
```

opens the sequential file "MGT" for sequential input. This enables you to retrieve information from the file (using INPUT# or LINE INPUT#). If "MGT" does not exist, a "File not found" error occurs.

See the chapter on "Disk Files" for programming information.

OPTION BASE

OPTION BASE *n*	**Statement**

Sets *n* as the minimum value for an array subscript.

N may be 1 or 0. The default is 0.

If you use this statement in a program, it must precede the DIM statement.

If the statement

```
OPTION BASE 1
```

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

OUT

OUT *port, data byte* **Statement**

Sends a *data byte* to a machine output *port.*

Port is an integer between 0 and 255. *Data byte* is also an integer between 0 to 255.

A port is an input/output location in memory. For information on assigned ports, see the Technical Reference Manual.

Example

```
OUT 32,100
```

sends 100 to port 32.

PEEK

PEEK(*memory location*) **Function**

Returns a byte from *memory location.*

The *memory location* must be in the range −32768 to 65535.

The value returned is an integer between 0 and 255. (For the interpretation of a negative value of *memory location*, see the statement VARPTR.)

PEEK is the complementary function of the statement POKE.

Example

```
A = PEEK (&H5A00)
```

POKE

POKE *memory location, data byte*	**Statement**

Writes *data byte* into *memory location.*

Both *memory location* and *data byte* must be integers. *Memory location* must be in the range −32768 to 65535.

POKE is the complementary statement of PEEK. The argument to PEEK is a memory location from which a byte is to be read.

PEEK and POKE are useful for storing data efficiently, loading assembly-language subroutines, and passing arguments (or results) to and from assembly-language subroutines.

For more information, see the Technical Reference Manual.

Example

```
10 POKE &H5A00, &HFF
```

POS

POS(*number*)	**Function**

Returns the position of the cursor.

Number is a dummy argument.

POS returns a number from 1 to 80 indicating the current cursor-column position on the display.

Example

```
PRINT TAB(40) POS(0)
```

prints 40. The PRINT TAB statement moves the cursor to position 40, therefore, POS(0) returns the value 40. (However, since a blank is inserted before the "4" to accommodate the sign, the "4" is actually at position 41).

Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) > 70 THEN IF A$ = CHR$(32)
    THEN A$ = CHR$(13)
190 PRINT A$;
200 PRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that you cannot correct mistakes). Your computer keyboard is the typewriter keyboard. The program will keep watch at the end of a line so that no word is divided between two lines.

PRINT

PRINT *data, . . .*	**Statement**

Prints numeric or string *data* on the display.

BASIC prints the values of the data items you list in this statement.

You may separate the data items by commas or semicolons. If you use commas, the cursor automatically advances to the next tab position before printing the next item. (BASIC divides each line into five tab positions, at columns 0, 16, 32, 48, and 64). If you use semicolons, it prints the items without any spaces between them.

A semicolon or comma at the end of a line causes the next PRINT statement to begin printing where the last one left off. If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

Single-precision numbers with six or fewer digits that can be accurately represented in ordinary (rather than exponential) format,

are printed in ordinary format. For example, 1E-7 is printed as .0000001; 1E-8 is printed as 1E-08.

Double-precision numbers with 16 or fewer digits that can be accurately represented in ordinary format, are printed using the ordinary format. For example, 1D-15 is printed as .000000000000001; 1D-16 is printed as 1D-16.

BASIC prints positive numbers with a leading blank. It prints all numbers with a trailing blank.

To insert strings into this statement, surround them with quotation marks.

Examples

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES";
"BETWEEN"; "THESE"; "WORDS"
```

prints on the display:
DONOTLEAVESPACESBETWEENTHESEWORDS

Sample Program

```
60 INPUT "ENTER THIS YEAR"; Y
70 INPUT "ENTER YOUR AGE";A
80 INPUT "ENTER A YEAR IN THE FUTURE";F
90 N = A + (F - Y)
100 PRINT "IN THE YEAR"F"YOU WILL BE"N"YEARS
    OLD"
RUN
```

Since F and N are positive numbers, PRINT inserts a space before and after them, therefore your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in line 100 by a comma,

```
100 PRINT "IN THE YEAR",F,"YOU WILL
    BE",N,"YEARS OLD"
```

BASIC would move to the next tab position after printing each data item.

PRINT USING

PRINT USING *format*; *data item*, . . .	**Statement**

Prints *data items* using a *format* specified by you.

Format consists of one or more field specifiers enclosed in quotes, or a string variable which contains the field specifier(s).

Data item may be string and/or numeric value(s).

This statement is especially useful for printing report headings, accounting reports, checks, or any other documents which require a specific format.

With PRINT USING, you may use certain characters (field specifiers) to format the field. These field specifiers are described below. They are followed by sample program lines and their output to the screen.

Specifiers for String Fields:

! Print the first character in the string only.

PRINT USING "!"; "PERSONNEL"
P

\spaces\ Print 2+ n characters from the string. If you type the backslashes without any spaces, BASIC prints two characters; with one space, BASIC prints three characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right. To enter a backslash, press (CLEAR)(?).

PRINT USING "\ƀƀƀ\"; "PERSONNEL"
(three spaces between the backslashes)
PERSO

& Print the string without modifications.

```
10 A$="TAKE":B$="RACE"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
TRACE
```

Specifiers for Numeric Fields:

#

Print the same number of digit positions as number signs (#). If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). Numbers are rounded as necessary. You may insert a decimal point at any position. In that case, the digits preceding the decimal point are always printed (as zero, if necessary).

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

```
PRINT USING "##.##";111.22
%111.22
```

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

```
PRINT USING "##.##";.75
0.75
```

```
PRINT USING "###.##";876.567
876.57
```

+

Print the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+##.##    ";
-98.45,3.50,22.22,-.9
-98.45    +3.50    +22.22    -0.90
```

```
PRINT USING "##.##+    ";
-98.45,3.50,22.22,-.9
98.54-    3.50+    22.22+    0.90-
```

(Note the use of spaces at the end of a format string to separate printed values).

−

Print a negative sign after negative numbers (and a space after positive numbers).

```
PRINT USING "###.#-"; -768.660
768.7-
```

**

Fill leading spaces with asterisks. The two asterisks also establish two more positions in the field.

```
PRINT USING "**####"; 44.0
****44
```

$$ Print a dollar sign immediately before the number. This specifies two more digit positions, one of which is the dollar sign.

PRINT USING "$$##.##"; 112.7890
$112.79

****$** Fill leading spaces with asterisks and print a dollar sign immediately before the number.

PRINT USING "**$##.##"; 8.333
***$8.33

, Print a comma before every third digit to the left of the decimal point. The comma establishes another digit position.

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

^ ^ ^ ^ Print in exponential format. The four exponent signs are placed after the digit position characters. To type the ^, press (CLEAR)(;). You may specify any decimal point position.

PRINT USING ".####^ ^ ^ ^"; 888888
.8889E+06

_ Print next character as a literal character.

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

Sample Program

```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P
470 CLS : PRINT "PAY TO THE ORDER OF  ";
480 PRINT USING "!! !!  "; F$; "."; M$; ".";
490 PRINT L$
500 PRINT :PRINT USING A$; P
```

In line 480, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the two spaces in "!!b!!b". These two spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for format and P for item list in line 500. Any serious use of the PRINT USING statement would probably require the use of variables at least for item list rather than constants. (We've used constants in our examples for the sake of better illustration).

When the program above is run, the output should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,435.60 DOLLARS
```

PRINT @

Statement

PRINT@ *location,*
PRINT@ (*row, column*),

Specifies exactly where printing is to begin.

The *location* specified must be a number between 0 and 1919. It can also be a a pair of numbers (*r, c*), where 23=>r=>0 and 79=>c=>0.

Whenever you instruct BASIC to PRINT @ the bottom line of the display, it generates an automatic line feed; everything on the display moves up one line. To suppress this automatic line feed, use a trailing semicolon at the end of the statement.

NOTE: If the string you are printing extends past column 80, BASIC prints the entire string on the next line.

Examples

```
PRINT @ (11,39), "*"
```

prints an asterisk in the middle of the display. The space between PRINT and @ is optional.

```
PRINT @ 0, "*"
```

prints an asterisk at the top left corner of the display.

PRINT TAB

PRINT TAB(*n*)	**Statement**

Moves the cursor to the *n* position on the current line.

TAB may be used more than once in a print list.

Since numeric expressions may be used to specify a TAB position, TAB can be very useful in creating tables, graphs of mathematical functions, etc.

TAB can't be used to move the cursor to the left. If the cursor is to the right of the specified position, the TAB statement is simply ignored.

The first parenthesis must be typed immediately after the word TAB.

If *n* is greater than 80, BASIC divides *n* by 80 and uses the remainder of the division as the tab position. For example, if you enter the line:

```
PRINT "NAME"; TAB(85); "AMOUNT"
```

BASIC converts TAB(84) into TAB(4). Since the cursor is already at column five after printing NAME, BASIC moves the string AMOUNT to the next line. If, instead, you had typed TAB(85), BASIC would print AMOUNT on the same line.

If the string you are printing is too long to fit on the current line, BASIC moves the string to the next line.

Example

```
PRINT TAB(5)  "TABBED 5"; TAB(25) "TABBED 25"
```

Notice that no punctuation is needed after the TAB modifiers.

Sample Program

```
220 CLS
230 PRINT TAB(2) "CATALOG NO."; TAB(16)
    "DESCRIPTION OF ITEM";
240 PRINT TAB(39) "QUANTITY"; TAB(51) "PRICE
    PER ITEM";
245 PRINT TAB(69) "TOTAL PRICE"
```

PRINT#

PRINT# *buffer, item1, item2, . . .*	**Statement**

Prints data *items* in a sequential disk file.

Buffer is the buffer number used to OPEN the file for input.

When you first OPEN a file for sequential output, BASIC sets a pointer to the beginning of the file — that's where PRINT# starts printing the values of the *items*. At the end of each PRINT# operation, the pointer advances, so values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to the display creates on the screen. For this reason, make sure to delimit the data so that it will be input correctly from the disk.

PRINT# does not compress the data before writing it to disk. It writes an ASCII-coded image of the data.

Examples

```
If A = 123.45
PRINT# 1,A
```

writes this nine-byte character sequence onto disk:

```
b123.45b carriage return
```

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example, if A = 2300 and B = 1.303, then

```
PRINT# 1, A,B
(ENTER)
```

writes the data on disk as

```
b 2300 bbbbbbbbbb 1.303b carriage return
```

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semicolons instead of commas.

Files can be written in a carefully controlled format using PRINT# USING. You can also use this option to control how many characters of a value are written to disk.

For example, suppose A$ = "LUDWIG", B$ = "VON", and C$ = "BEETHOVEN". Then the statement

```
PRINT# 1, USING"!.!.\bb\";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET
```

(In this case, we didn't want to add any explicit delimiters.) See PRINT USING for more information on the USING option.

PUT

PUT *buffer* [,*record*]	**Statement**

Puts a *record* in a direct-access disk file.

Buffer is the same buffer used to OPEN the file.

Record is the record number you want to PUT into the file. It is an integer between 1 and 65535. If omitted, the current record number is used.

This statement moves data from the buffer of a file into a specified place in the file.

If *record* is higher than the end-of-file record number, then *record* becomes the new end-of-file record number.

The first time you use PUT after OPENing a file, you must specify the *record*. The first time you access a file via a particular buffer, the next record is set equal to one. (The next record is the record whose number is one greater than the last record accessed).

See the chapter on "Disk Files" for programming information.

```
PUT 1
```

writes the next record from buffer 1 to a direct-access file.

```
PUT 1, 25
```

writes record 25 from buffer 1 to a direct-access file.

RANDOM

RANDOM	Function

Reseeds the random number generator.

If your program uses the RND function, every time you load it, BASIC generates the same sequence of pseudorandom numbers. Therefore, you may want to put RANDOM at the beginning of the program. This will help ensure that you get a different sequence of pseudorandom numbers each time you run the program.

RANDOM needs to execute just once.

Sample Program

```
600 CLS : RANDOM
610 INPUT "PICK A NUMBER BETWEEN 1 AND 5"; A
620 B = RND(5)
630 IF A = B THEN 650
640 PRINT "YOU LOSE, THE ANSWER IS" B "--TRY
    AGAIN."
645 GOTO 610
650 PRINT "YOU PICKED THE RIGHT NUMBER -- YOU
    WIN!": GOTO 610
```

READ

READ *variable, ...*	**Statement**

Reads values from a DATA statement and assigns them to *variables*.

BASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is used; the second time, the second value is used, and so on.

A single READ may access one or more DATA statements (each DATA statement is accessed in order), or several READs may access the same DATA statement.

The values read must agree with the variable types specified in list of variables, otherwise, a "Syntax error" occurs. If the number of variables in the READ statement exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed.

If the number of variables specified is lower than the number of elements in the DATA statement(s), subsequent READ statements begin reading data at the first unread element.

Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable "T".

Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40 PRINT "NAME", "AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$, AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERS, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, "END"
```

REM

REM	Statement

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line. This allows you to insert remarks into your program for documentation. Then, when you look at a listing of your program, or someone else does, it will be easier to figure it out.

If REM is used in a multi-statement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for REM.

Sample Program

```
110 DIM V(20)
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
```

OR

```
110 DIM V(20)
120 FOR I=1 TO 20      'CALCULATE AVERAGE VELOCITY
130 SUM=SUM + V(I)
140 NEXT I
```

RENUM

RENUM [***new line***] [,[***line***] [,***increment***]]	**Statement**

Renumbers a program, starting at *line*, using *new line* as the first new line and *increment* for the new sequence.

If you omit *new line*, BASIC starts numbering at line 10. If you omit the *line*, it renumbers the entire program. If you omit *increment*, it jumps 10 numbers between lines.

RENUM also changes all line number references appearing after GOTO, GOSUB, THEN, ELSE, ON . . . GOTO, ON . . . GOSUB, ON ERROR GOTO, RESUME, and ERL[relational operator].

Examples

```
RENUM
```

renumbers the entire resident program, incrementing by 10's. The new number of the first line will be 10.

```
RENUM 600, 5000, 100
```

renumbers all lines numbered from 5000 up. The first renumbered line will become 600, and an increment of 100 will be used between subsequent lines.

```
RENUM 10000, 1000
```

renumbers line 1000 and all higher-numbered lines. The first renumbered line will become line 10000. An increment of 10 will be used between subsequent line numbers.

```
RENUM 100, , 100
```

renumbers the entire program, starting with a new line number of 100, and incrementing by 100's. Notice that the commas must be retained even though the middle argument is gone.

Error Conditions

1. RENUM cannot be used to change the order of program lines. For example, if the original program has lines numbered 10, 20 and 30, then the command:

   ```
   RENUM 15, 30
   ```

is illegal, since the result would be to move the third line of the program ahead of the second. In this case, an "Illegal function call" error occurs, and the original program is left unchanged.

2. RENUM will not create new line numbers greater than 65529. Instead, an "Illegal function call" error occurs, and the original program is left unchanged.
3. If an undefined line number is used inside your original program, RENUM prints a warning message, Undefined line XXXX in YYYY", where XXXX is the original line number reference and YYYY is the original number of the line containing XXXX. Note that RENUM renumbers the program in spite of this warning message. It does not change the incorrect line number reference, but it does renumber YYYY, according to the parameters in your RENUM command.

RESTORE

RESTORE [***line***]	**Statement**

Restores a program's access to previously-read DATA statements.

This lets your program re-use the same DATA lines.
If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA THIS IS THE FIRST ITEM, AND THIS IS
    THE SECOND
```

When this program is run,

```
THIS IS THE FIRST ITEM    THIS IS THE FIRST ITEM
```

is printed on the display. Because of the RESTORE statement in line 170, the second READ statement starts over with the first DATA item.

RESUME

RESUME [*line*]
RESUME NEXT

Statement

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME 0 both cause the computer to return to the statement in which the error occurred.

RESUME *line* causes the computer to branch to the specified line number.

RESUME NEXT causes the computer to branch to the statement following the point at which the error occurred.

A RESUME that is not in an error-handling routine causes a "RESUME without error" message.

Examples

```
RESUME
```

if an error has occurred, this line transfers program control to the statement in which it occurred.

```
RESUME 10
```

if an error has occurred, transfers control to line 10.

Sample Program

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230) AND(ERL=90) THEN PRINT "TRY
    AGAIN" : RESUME 80
```

RETURN

RETURN	**Statement**

Returns control to the line immediately following the most recently executed GOSUB.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A
    CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS"; R
350 GOSUB 370
360 PRINT "AREA IS" ; A: END
370 A = 3.14 * R * R
380 RETURN
```

RIGHT$

RIGHT$(*string, number*)	**Function**

Returns the rightmost *number* characters of *string*.

RIGHT$ returns the last *number* characters of *string*. If LEN (string) is less than or equal to *number*, the entire string is returned.

Examples:

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

Sample Program

```
850 RESTORE : ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT  RIGHT$(COMPANY$, 2), : GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMON MANUFACTURING COMPANY,
    HAMMOND, IN"
```

This program prints the name of the state in which each company is located.

RND

RND(*number*)	**Function**

Generates a pseudorandom number between 0 and *number*.

Number must be greater than or equal to 0 and less than 32768.

RND produces a pseudorandom number using the current "seed" number. BASIC generates the seed internally, therefore, it is not accessible to the user. RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a single-precision value between 0 and 1, RND(number) returns an integer between 1 and *number*. For example, RND(55) returns a pseudorandom integer between 1 and 55. RND(55.5) returns a pseudorandom number between 1 and 56 (the argument is rounded).

Examples

```
A = RND(2)
```

assigns A a value of 1 or 2.

```
A = RND(45)
```

assigns A a random integer between 1 and 45.

```
PRINT RND (0)
```

prints a decimal fraction between 0 and 1.

ROW

ROW(*number*)	Function

Returns the row position of the cursor.

Number is a dummy argument.

ROW finds the row in which the cursor is currently located and returns that row number. The 24 rows are numbered 0-23.

Examples

```
X = ROW(Y)
```

assigns the cursor's current row number to X.

Sample Program

When you type a key, the program below prints: the keyboard character, the cursor's row number and column number, and the character's ASCII code.

```
100 CLS
110 R=0: C=0
120 PRINT@(21,32), "ROW", "COLUMN"
130 X$ = INPUT$(1)
140 PRINT @(R,C), X$;
150 C=POS(0): R=ROW(0)
160 PRINT @ (22,32),R,C;
163 PRINT @ (23,32), STRING$(20,32);
165 PRINT @(23,32), "ASCII CODE IS
    "HEX$(ASC(X$));
170 PRINT @ (R,C),"";
180 GOTO 130
```

RSET

Statement

RSET *field name* = *data*

Sets *data* in a direct-access buffer *field name*.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

RUN

Statement

RUN [***line***]
RUN *filespec*[**,R**]

Runs a program.

RUN followed by a *line* or nothing at all simply executes the program in memory, starting at *line* or at the beginning of the program.

RUN followed by a *filespec* loads a program from disk and then runs it. Any resident BASIC program is replaced by the new program.

Option R leaves all previously OPEN files open. If omitted, BASIC closes all open files.

RUN automatically CLEARS all variables. However, it does not re-set the value of an ERL variable.

Examples

```
RUN
```

starts execution at lowest line number.

```
RUN 100
```

starts execution at line 100.

```
RUN "PROGRAM/A"
```

loads and executes PROGRAM/A.

```
RUN "EDITDATA", R
```

loads and executes EDITDATA, leaving OPEN files open.

SAVE

SAVE *"filespec"* [,**A**] [,**P**]	**Statement**

Saves a program in a disk file under *filespec*.

If *filespec* already exists, its contents will be lost as the file is re-created.

SAVE without the A option saves the program in a compressed format. This takes up less disk space. It also helps in performing SAVEs and LOADs faster. BASIC programs are stored in RAM using compressed format.

Using the A option causes the program to be saved in ASCII format. This takes up more disk space. However, the ASCII format allows you to MERGE this program later on. Also, data programs which will be read by other programs must usually be in ASCII.

For compressed-format programs, a useful convention is to use the extension BAS. For ASCII-format programs, use /TXT.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it fails. The only operations that can be performed on a protected file are: RUN, LOAD, MERGE, and CHAIN.

Examples

```
SAVE  "FILE1/BAS.JOHNQDOE:3"
```

saves the resident BASIC program in compressed format. The file name is FILE1; the extension is /BAS; the password is JOHNQDOE. The file is placed on Drive 3.

```
SAVE "MATHPAK/TXT", A
```

saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non-write-protected drive.

SGN

SGN(*number*)	**Function**

Determines *number's* sign.

If *number* is a negative number, SGN returns −1. If *number* is a positive number, SGN returns 1. If *number* is zero, SGN returns 0.

Examples

```
Y = SGN(A * B)
```

determines what the sign of the expression A * B is, and passes the appropriate number (−1,0,1) to Y.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SIN

SIN(*number*) **Function**

Computes the sine of *number*.

Number must be in radians. To obtain the sine of *number* when *number* is in degrees, use SIN(number * .01745329). The result is always single precision.

Examples

```
PRINT SIN(7.96)
```

prints .994385.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN A * .01745329)
```

SOUND

SOUND *tone, duration* **Statement**

Generates a sound with the *tone* and *duration* specified.

Tone is a digit between 0 and 7. It specifies the sound's frequency level. Zero specifies the lowest frequency level; seven specifies the highest.

Duration is an integer between 0 and 31. It specifies for how long the sound is to be generated. Zero specifies the shortest duration; 31 the longest.

This statement can be especially useful in educational applications. For example, you can have the computer respond with a sound if a

user has answered a program's prompt incorrectly (or vice versa).

Sample Program

```
10 INPUT "IN HONOR OF WHOM WAS THE CONTINENT OF
   AMERICA NAMED"; A$
20 IF A$="AMERIGO VESPUCCI" THEN SOUND 7,2 ELSE
   GOTO 40
30 PRINT "THAT'S RIGHT!": END
40 SOUND 1,2 : PRINT "THE CORRECT ANSWER IS
   AMERIGO VESPUCCI"
```

SPACE$

SPACE$(*number*)	**Function**

Returns a string of *number* spaces.

Number must be in the range 0 to 255.

Example

```
PRINT "DESCRIPTION" SPACE$(4) "TYPE" SPACE$(9)
"QUANTITY"
```

prints DESCRIPTION, four spaces, TYPE, nine spaces, QUANTITY.

Sample Program

```
920 PRINT "Here"
930 PRINT SPACE$(13) "is"
940 PRINT SPACE$(26) "an"
950 PRINT SPACE$(39) "example"
960 PRINT SPACE$(52) "of"
970 PRINT SPACE$(65) "SPACE$"
```

SPC

SPC(*number*)	**Function**

Prints a line of *number* blanks.

Number is in the range 0 to 255. SPC does not use string space. The left parenthesis must immediately follow SPC.

SPC may only be used with PRINT, LPRINT, or PRINT#.

Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints HELLO, 15 spaces, THERE

SQR

SQR(*number*)	**Function**

Calculates the square root of *number*.

The *number* must be greater than zero.

The result is always single precision.

Example

```
PRINT SQR(155.7)
```

prints 12.478.

Sample Program

```
680 INPUT "TOTAL RESISTANCE (OHMS)"; R
690 INPUT "TOTAL REACTANCE (OHMS)"; X
700 Z = SQR((R * R) + (X * X))
710 PRINT "TOTAL IMPEDANCE (OHMS) IS" Z
```

This program computes the total impedance for series circuits.

STOP

STOP	Statement

Stops program execution.

When a program encounters a STOP statement, it prints the message BREAK IN, followed by the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

The CONT command resumes execution at the point it was halted. But if the program itself is altered during the break, CONT cannot be used.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number between 1 and 10 is assigned to X, then program execution halts at line 2270. You can now examine the value X with PRINT X. Type CONT to start the cycle again.

STR$

STR$(*number*) **Function**

Converts *number* into a string.

If *number* is positive, STR$ places a blank before the string.

While arithmetic operations may be performed on *number*, only string functions and operations may be performed on the string.

Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S$.

Sample Program

```
10 A = 1.6 : B# = A : C# = VAL(STR$(A))
20 PRINT "REGULAR CONVERSION" TAB(40) "SPECIAL
   CONVERSION"
30 PRINT B# TAB(40) C#
```

STRING$

STRING$(*number,character*) **Function**

Returns a string of *number* characters.

Number must be in the range 0 to 255.

Character is a string or an ASCII code. If you use a string constant, it must be enclosed in quotes. All the characters in the string will have either the ASCII code specified, or the first letter of the string specified.

STRING$ is useful for creating graphs or tables.

Examples:

```
B$ = STRING$(25, "X")
```

puts a string of 25 "X"s into B$.

```
PRINT STRING$(50, 10)
```

prints 50 blank lines on the display, since 10 is the ASCII code for a line feed.

Sample Program

```
1040 CLEAR 300
1050 INPUT "TYPE IN THREE NUMBERS BETWEEN 33
     AND 159"; N1, N2, N3
1060 CLS: FOR I = 1 TO 4: PRINT STRING$(20,
     N1): NEXT I
1070 FOR J = 1 TO 2: PRINT STRING$(40, N2):
     NEXT J
1080 PRINT STRING$(80, N3)
```

This program prints three strings. Each string has the character corresponding to one of the ASCII codes provided.

SWAP

SWAP ***variable1, variable2***	**Statement**

Exchanges the values of two variables.

Variables of any type may be SWAPped (integer, single precision, double precision, string). However, both must be of the same type, otherwise, a "Type mismatch" error results.

Either or both of the variables may be elements of arrays. If one or both of the variables are non-array variables which have not been assigned values, an "Illegal Function Call" error results.

Example

```
SWAP F1#, F2#
```

swaps the contents of F1# and F2#. The contents of F2# are put into F1#, and the contents of F1# are put into F2#.

Sample Program

```
10 A$="ONE ":B$="ALL ":C$="FOR "
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
ONE FOR ALL
ALL FOR ONE
```

SYSTEM

Statement

SYSTEM [*"command"*]

Returns you to TRSDOS level.

Command tells the system to execute the specified TRSDOS command and immediately return to BASIC. Your program and variables are not affected. If *command* is a constant, it must be enclosed in quotes. You can specify only the TRSDOS library commands, not the utilities.

If you omit *command*, SYSTEM returns to the TRSDOS Ready mode. Your resident BASIC program is not retained in memory.

NOTE: You cannot call DEBUG from BASIC.

Examples

```
SYSTEM
```

returns you to TRSDOS. Your resident BASIC program is lost.

```
SYSTEM "DIR"
```

runs the TRSDOS command, DIR (print directory), then returns to BASIC. Your resident BASIC program remains intact.

TAB

TAB(*number*) **Function**

Spaces to position *number* on the display.

Number must be in the range 1 to 255.

If the current print position is already beyond space *number*, TAB goes to that position on the next line. Space one is the leftmost position; the width minus one is the rightmost position.

TAB may only be used with the PRINT and LPRINT statements.

Sample Program

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
RUN
```

The display shows:

```
NAME                    AMOUNT

G.T.JONES               $25.00
```

TAN

TAN(*number*) **Function**

Computes the tangent of *number*.

Number must be in radians. To obtain the tangent of *number* when it is in degrees, use TAN (number * .01745329). The result is always single precision.

Examples

```
PRINT TAN(7.96)
```

prints −9.39702.

Sample Program

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

TIME$

TIME$	**Function**

Returns the time of the day.

This function lets you use the time in a program.

The operator sets the time initially when TRSDOS is started up. When you request the time, TIME$ supplies it using this format:

```
14:47:18
```

which means 14 hours, 47 minutes and 18 seconds (24-hour clock).

To change the time, use the TRSDOS command, TIME. For example,

```
SYSTEM "TIME 10:15:00"
```

Example

```
A$ = TIME$
```

stores the current time in A$.

Sample Program

```
1130 SYSTEM "TIME 10:15:00"
1140 IF LEFT$(TIME$, 5) = "10:15" THEN PRINT
     "Time is 10:15 A.M.--time to pick up the
     mail." : END
1150 GOTO 1140
```

TROFF, TRON

	Statements
TROFF **TRON**	

Turn the "trace function" on/off.

The trace function lets you follow program flow. This is helpful for debugging and analyzing of the execution of a program.

Each time the program advances to a new line, TRON displays that line number inside a pair of brackets. TROFF turns the tracer off.

Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

Lines 2290 and 2310 above might be helpful in assuring you that line 2300 is actually being executed, since each time it is executed [2300] is printed on the display.

After a program is debugged, the TRON and TROFF statements can be removed.

USR

	Function
USR[*digit*](*expression*)	

Calls a user's assembly-language subroutine identified with *digit* and passes *expression* to that subroutine.

The *digit* you specify must correspond to the *digit* supplied with the DEF USR statement for that routine. If *digit* is omitted, zero is assumed.

This function lets you call as many as 10 machine-language subroutines, then continue execution of your BASIC program. Subroutines must have been previously defined with DEF USR[digit] statements.

When BASIC encounters a USR call, it transfers control to the address defined in the DEF USR[digit] statement. (This address specifies the entry point to your machine-language subroutine.)

"Machine language" is the low-level language used internally by your computer. It consists of Z-80 microprocessor instructions. Machine-language subroutines are useful for special applications (things you can't do in BASIC) and for doing things very fast (like to "white-out" the display).

Writing such routines requires familiarity with assembly-language programming and with the Z-80 instruction set. There are books available on this subject; check your local Radio Shack or a book store.

Example

```
X = USR5(Y)
```

calls the machine-language routine USR5, previously defined in a DEF USR5 = *address* statement.

Passing arguments from BASIC to the subroutine:

Upon entry to a USR subroutine, the following register contents are set up (for notation, see the TRSDOS reference section in this manual):

A = Type of argument in USR[digit] reference
A = 8 if argument is double-precision
A = 4 if argument is single-precision

		A = 2 if argument is integer A = 3 if argument is string
HL	=	When the argument is a number, this register points to the argument storage area(ASA) described later.
DE	=	When the argument is a string, this register points to a string description, as follows: The first byte gives the length of the string. The next two bytes give the address where the string is stored: least significant byte (LSB) followed by most significant byte(MSB).

Description of Argument Storage Area (ASA) — for numeric values only.

For double-precision numbers:

ASA + 3	Exponent in 128-excess form, e.g., a value of 128 indicates a 0 exponent; a value of 66 indicates a −62 exponent. A value of 0 always indicates the number is zero.
ASA + 2	Highest seven bits of the mantissa with hidden (implied) leading one. Bit 7 is the sign of the number(0 positive, 1 negative), e.g., a value of X'84' indicates the number is negative and the MSB of the mantissa is X'84'. A value of X'04' indicates the number is positive and the MSB of the mantissa is X'84'.
ASA + 1	Next MSB of the mantissa.
ASA	Next MSB.
ASA - 1	Next MSB.
ASA - 2	Next MSB.
ASA - 3	Next MSB.
ASA - 4	Lowest eight bits of the mantissa.

For single-precision numbers:

ASA	LSB of the mantissa.
ASA + 1 through ASA + 3	Same as for double-precision numbers.

For integers:

ASA	LSB of the number
ASA + 1	MSB of the number. Together, the two bytes represent the number in signed, two's complement form.

Your routine can call BASIC's FRCINT routine to put the argument into HL in 16-bit, signed two's complement form. The address of FRCINT is stored in [X'2603', X'2604'].

For example, you can put the following code at the beginning of your subroutine:

```
FRCINT  EQU     2603H           ;CONVERTS USR ARGUMENT
                                ;TO INTEGER IN HL
        LD      HL,CTNU         ;(HL)=CONTINUATION
                                ;ADDRESS
        PUSH    HL              ;SAVE IT FOR RETURN
                                ;FROM FRCINT
        LD      HL,(FRCINT)     ;(HL)=FORCE INTEGER
                                ;ROUTINE
        JP      (HL)            ;DO FRCINT ROUTINE
```

Returning values from the subroutine to BASIC:

If the USR[digit] expression is a variable, you can modify its value by changing the ASA or string contents, as pointed to by HL or DE. For example, the statement:

```
X=USR1(A%)
```

transfers control to the USR1 subroutine, with HL pointing to the two-byte ASA for integer variable A%. Suppose you modify the contents of its storage area. When you do a RET instruction to return to BASIC, A% will have a new value, and X will be assigned this new value.

In general, USR[digit](expression) will return the same type of value as the expression. However, you can use BASIC's MAKINT routine to return an integer value. The address of the MAKINT routine is stored at [X'2605',X'2606'].

For example, you might include the following code at the end of your program to return a value to BASIC:

```
MAKINT  EQU     2605H
        LD      HL,VAL          ;VAL IS THE VALUE TO
                                ;BE RETURNED.
        PUSH    HL              ;SAVE VALUE IN STACK
        LD      HL, (MAKINT)    ;RESTORE VAL INTO HL
        EX      (SP), HL        ;AND PUT MAKINT
                                ;INTO STACK
        RET
```

VAL

VAL(*string*)	**Function**

Calculates the numerical value of *string*.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

For example, if A$ = "12" and B$ = "34" then VAL(A$ + "." + B$) returns the value 12.34 and VAL(A$ + "E" + B$) returns the value 12E34, that is, 12 * 10^34.

VAL terminates its evaluation on the first character which has no meaning in a numeric value.

If the string is non-numeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 1.234E+08.

```
B = VAL("3" + "*" + "2")
```

assigns the value 3 to B (the asterisk has no meaning in a numeric term).

Sample Program

```
10 READ NAMES$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699
   THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) > 90801 AND VAL(ZIP$) <= 90815
   THEN PRINT NAME$ TAB(25) "LONG BEACH"
```

VARPTR

Function

VARPTR (*variable*)
or
VARPTR (*#buffer*)

Returns the absolute memory address.

VARPTR can help you locate a value in memory. When used with *variable*, it returns the address of the first byte of data identified with *variable*.

When used with *buffer*, it returns the address of the file's data buffer.

If the *variable* you specify has not been assigned a value, an "Illegal Function Call" occurs. If you specify a *buffer* that was not allocated when loading BASIC, a "Bad file number" error occurs. (See Chapter 1 for information on how to load BASIC.)

VARPTR is used primarily to pass a value to a machine-language subroutine via USR[digit]. Since VARPTR returns an address which indicates where the value of a variable is stored, this address can be passed to a machine-language subroutine as the argument of USR; the subroutine can then extract the contents of the variable with the help of the address that was supplied to it.

If VARPTR returns a negative address, add it to 65536 to obtain the actual address.

If VARPTR(integer variable) returns address K:

Address K contains the least significant byte (LSB) of the 2-byte integer.

Address K + 1 contains the most significant byte (MSB) of the integer.

If VARPTR(single-precision variable) returns address K:

(K)*	=	LSB of value
(K + 1)	=	Next most significant byte(Next MSB)
(K + 2)	=	MSB with hidden (implied) leading one. Most significant bit is the sign of the number
(K + 3)	=	exponent of value excess 128(128 is added to the exponent).

*(K) signifies "contents of address K"

If VARPTR(double-precision variable) returns K:

(K)	= LSB of value
(K + 1)	= Next MSB
(K + . . .)	= Next MSB
(K + 6)	= MSB with hidden (implied) leading one. Most significant bit is the sign of the number.
(K + 7)	= exponent of value excess 128 (128 is added to the exponent).

For single and double-precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. It is set to 0 if the number is positive or to 1 if the number is negative. See examples below.

If VARPTR(string variable) returns K:

(K)	= length of string
(K + 1)	= LSB of string value starting address
(K + 2)	= MSB of string value starting address

The address will probably be in high RAM where string storage space has been set aside. But, if your string variable is a constant (a string literal), then it will point to the area of memory where the program line with the constant is stored, in the program buffer area. Thus, program statements like A$="HELLO" do not use string storage space.

For all of the above variables, addresses (K-1) and (K-2) stores the TRS-80 Character Code for the variable name. Address (K-3) contains a descriptor code that tells the computer what the variable type is. Integer is 02; single precision is 04; double precision is 08; and string is 03.

VARPTR(array variable) returns the address for the first byte of that element in the array. The element consists of 2 bytes if it is an integer array; 3 bytes if it is a string array; 4 bytes if it is a single precision array; and 8 bytes if it is a double precision array.

The first element in the array is preceded by:

1. A sequence of two bytes per dimension, each two-byte pair indicating the "depth" of each respective dimension.
2. A single byte indicating the total number of dimensions in the array.
3. A two-byte pair indicating the total number of elements in the array.
4. A two-byte pair containing the ASCII-coded array name.

5. A one-byte type-descriptor(02 = Integer, 03=String, 04 = Single=Precision, 08 = Double-Precision).

Item 1 immediately precedes the first element, Item 2 precedes Item 1, and so on.

The elements of the array are stored sequentially with the first dimension-subscripts varying "fastest", then the second, etc.

Examples

A! = 2 is stored as follows:

2 = 10 Binary, normalized as .1E2 = .1 × 10 (to the second)
So exponent of A is 128+2 = 130 (called excess 128)
MSB of A is 10000000; however, the high bit is changed to zero since the value is positive(called hidden or implied leading one).

So A! is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
130	0	0	0

A!= -.5 is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
128	128	0	0

A!=7 is stored as

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
131	96	0	0

A!= -7:

Exponent(K + 3)	MSB(K + 2)	Next MSB(K + 1)	LSB(K)
131	224	0	0

Zero is stored as a zero-exponent. The other bytes are insignificant.

```
Y = USR1(VARPTR(number))
```

If number is an integer value, VARPTR(number) finds the address of the least significant byte of number. This address is passed to the subroutine, which in turn passes its result to Y.

WAIT

Statement

WAIT ***port, integer1*** [***,integer2***]

Suspends program execution until a machine input *port* develops a specified bit pattern. (A port is an input/output location.)

The data read at the port is exclusive OR'ed with *integer2*, then AND'ed with *integer1*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *integer2* is omitted, it is assumed to be zero.

It is possible to enter an infinite loop with the WAIT statement. In this case, you will have to manually restart the machine. To avoid this, WAIT must have the specified value at port number during some point in program execution.

For information on assigned ports, refer to the Technical Reference Manual.

Example

```
100 WAIT 32,2
```

WHILE . . . . WEND

Statement

WHILE *expression*

.
.
.
.
{loop statements}
.
WEND

Execute a series of statements in a loop as long as a given condition is true.

If *expression* is not zero (true), BASIC executes loop statements until it encounters a WEND. BASIC returns to the WHILE statement and checks *expression*. If it is still true, BASIC repeats the process. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND causes a "WEND without WHILE" error.

Sample Program

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115 FLIPS=0
120 FOR I=1 TO J-1
130 IF A$(I)>A$(I+1)THEN SWAP A$(I), A$(I+1):
    FLIPS=1
140 NEXT I
150 WEND
```

This program sorts the elements in array A$. Control falls out of the WHILE loop when no more SWAPS are performed on line 130.

WRITE

Statement

WRITE [*data, . . .*]

Writes *data* on the display.

WRITE prints the values of the data items you type. If *data* is omitted, BASIC prints a blank line. The *data* may be numeric and/or string. They must be separated by commas.

When the *data* are printed, each data item is separated from the last by a comma. Strings are delimited by quotation marks. After printing the last item on the list, BASIC inserts a carriage return.

Example

```
10 D=95:B=76:V$="GOOD BYE"
20 WRITE D, B, V$
RUN
 95, 76, "GOOD BYE"
Ready
```

WRITE#

WRITE# *buffer, data, . . .*	**Statement**

Writes *data* to a sequential-access file.

Buffer must be the number used to OPEN the file.

The *data* you enter may be numeric or string expressions.

WRITE# inserts commas between the data items as they are written to disk. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

The items on *data* must be separated by commas.

WRITE# inserts a carriage return after writing the last data item to disk.

For example, if

```
A$="MICROCOMPUTER" and B$="NEWS"
```

the statement

```
WRITE#1, A$,B$
```

writes the following image to disk:

```
"MICROCOMPUTER","NEWS"
```