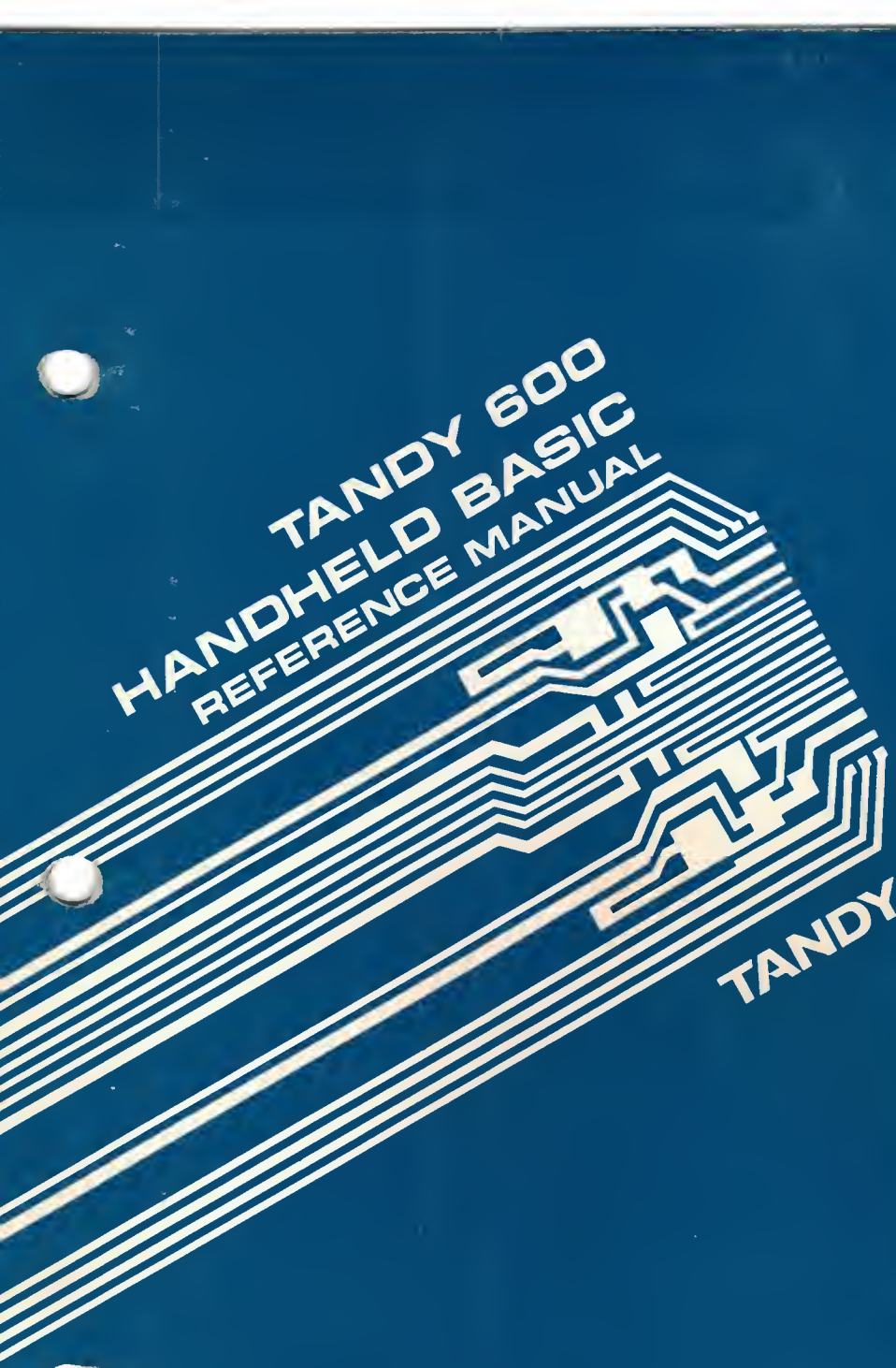
TANDY 600 HANDHELD BASIC REFERENCE MANUAL



TANDY®

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

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment. RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. **This warranty is only applicable to purcheses of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE." NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.**

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. SOFTWARE LICENSE

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

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on a multiuser or network system only if either, the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack.

VI. STATE LAW RIGHTS

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

6/86

TANDY® 600

HANDHELD BASIC REFERENCE MANUAL

10 9 8 7 6 5 4 3 2

Contents

INTRODUCTION TO BASIC

About This Manual

This manual describes Handheld BASIC for the Tandy 600. It is a reference manual, not a tutorial. We assume you already know BASIC and are using this manual to locate information quickly. If you do not know BASIC, see your Radio Shack dealer for the following book:

Learning BASIC for the Tandy 2000/1000
by David Lien, Cat. No. 25-1500

Your local bookstore has many books about BASIC available that are written in tutorial fashion.

Notations

The following notations are used throughout this manual:

CAPITALS	Material that you must enter exactly as it appears.
italics	Words, letters, characters, or values within command lines you must supply from a set of acceptable entries. Elsewhere, italics are used for emphasis.
. . . (ellipsis)	Items preceding the ellipsis may be repeated.
[ ]	Items enclosed in brackets are optional.
&H*nnnn*	*nnnn* is a hexadecimal number.
&O*nnnnn*	*nnnnn* is an octal number.
[keyname]	A key on your keyboard.
␣	A blank character (ASCII code 32). For example, in BASIC␣␣PROG two spaces are between BASIC and PROG.

Terms

The following terms are used in this manual:

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

Chapter 1

ABOUT HANDHELD BASIC

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

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

- Easier machine language subroutine access.
- Use of application program files from within BASIC programs.
- More accurate math operations with Binary Coded Decimal calculation system.

Naming Files

When you create a work file for an application, the first thing you do is enter a filename for the new file. Valid filenames use the following format: *filename.ext*

filename — a name you choose to identify the file

Type 8 or fewer letters and/or numbers, including special symbols: $ & # % ' ( ) @ ^ { } ! No spaces are allowed.

needed only when an extension follows

extension — a three character extension to identify the application

Type 3 or fewer letters and/or numbers, including special symbols: $ & # % ' ( ) @ ^ { } ! No spaces are allowed.

When you are creating a file within an application, the program automatically adds the appropriate extension. Refer to "Work File Extensions" for the proper extensions for each application's data files.

Device Names

BASIC uses device identifiers (dev:) to indicate a physical device to be used for communication. These names are:

KYBD: keyboard. Use for input only.
SCRN: screen. Use for output only.
LPT1: printer. Use for output only.

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

INSERTING AND USING THE BASIC ROM

On the bottom of the Tandy 600 is a covered compartment which contains the ROM chips that control your computer. The Tandy 600 comes with 5 ROM chips. The first 4 (from left to right) contain the operating system and all applications and functions except Multiplan.

The fifth chip, on the far right, contains the Multiplan application. The Multiplan chip is encased in a Molex carrier, which facilitates easy removal and insertion of the chip.

To insert the Handheld BASIC ROM chip in your Tandy 600, you must exchange the BASIC ROM chip for the Multiplan chip.

> **WARNING: Static and other magnetic fields can wipe out your ROM chips. Never touch the pins of the Tandy 600 ROM chips. Note that the last ROM slot must always contain a ROM chip, whether it is the Multiplan chip, the BASIC chip, or any chip made available in the future. Never remove any of the first four ROM chips.**

To exchange the BASIC ROM chip for the Multiplan chip, first save all your data files onto disk, then turn off the computer and close the viewing screen. Turn over the computer so that the removable panel is at the lower left corner as you face the computer. Pull up the panel from the notched lower side.

The Multiplan ROM is the chip on the far right of the enclosure. To remove the Multiplan ROM, use the top and bottom extensions on the Molex carrier to gently lift the chip and carrier from its socket. Now, insert the BASIC chip and carrier into the socket. The Molex carrier/chip assembly must be inserted with the 2 holes in the carrier at the top of the socket. (See the illustration on the following page.)

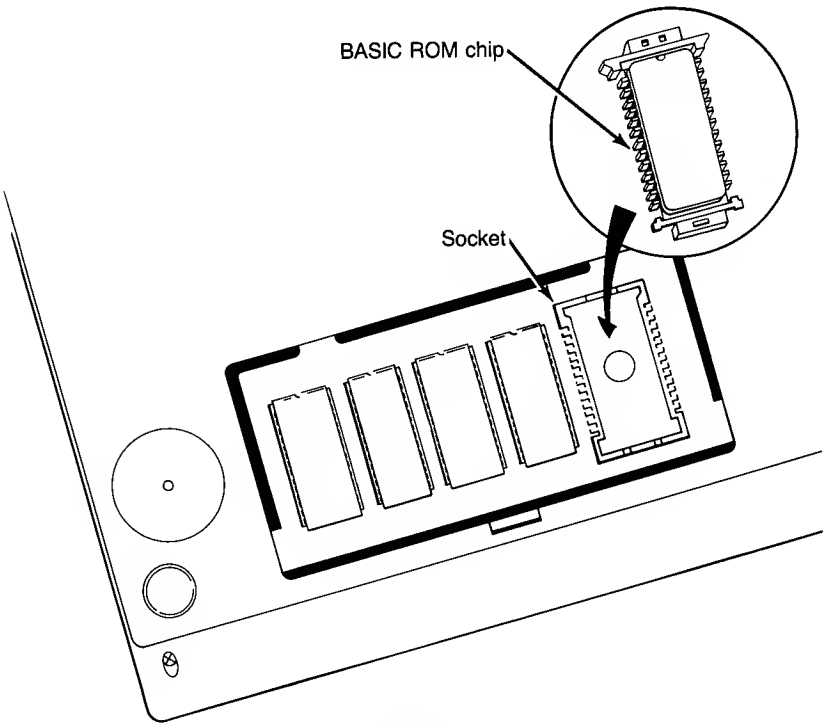


FIGURE A

When you turn on the computer and the Tandy 600 Screen menu reappears, BASIC is now listed as the last option in the command menu. To enter BASIC, move the cursor over the BASIC listing, and press [ENTER] twice. When the OK prompt appears, you are ready to begin programming in BASIC.

> **Note:** If your Tandy 600 has 128K of memory or more, you may save BASIC onto a diskette using the COPY command at the system manager level. This lets you load and use BASIC while keeping the Multiplan ROM in place and eliminates the need for swapping application ROM chips.

Chapter 3

TYPING AND SAVING BASIC PROGRAMS

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

The main steps in running a program are:

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

Entering BASIC

Position the cursor over the BASIC file in the Command Menu, and press [ENTER] twice. When the cursor flashes below the OK prompt, you are ready to begin programming in BASIC.

Typing the Program

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

```
10 A$="MASTERING BASIC " [ENTER]
15 B$="IS A SIMPLE PROCESS" [ENTER]
20 PRINT A$; B$ [ENTER]
```

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

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

Tell BASIC to execute this program by typing:

```
RUN [ENTER]
```

Your screen should display:

```
MASTERING BASIC IS A SIMPLE PROCESS
```

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

Saving the Program

You can save any BASIC program in RAM or on disk by assigning it a *filename*. The filename must be enclosed in quotation marks.

For example, to save the program we just wrote on the Disk Drive, with the filename *test.bas*, use the following command:

```
SAVE "A:test.bas" [ENTER]
```

You can also save the file in RAM with this command:

```
SAVE "test.bas"
```

which saves the program as *test.bas* in RAM and displays the filename on the menu when you exit BASIC.

After the program is saved in RAM or on the disk, BASIC displays its Ok prompt.

> **Note:** If you do not assign a filename to your program when you exit BASIC, the computer automatically saves the program under the filename WORK.BMI. When you re-enter BASIC and do not specify a filename at the "file to load" prompt, the computer loads the contents of WORK.BMI into memory.

Loading the Program Into Memory

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

For example, to load the program *test.bas* from disk, type:

```
LOAD "A:test.bas"
```

Another way to load and run a program saved in RAM is to type:

RUN "*filename*"

RUN automatically loads and runs the program in RAM specified by *filename*.

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

> **Note:** Whenever you run a BASIC program, the computer automatically creates a duplicate .BMI (BASIC Mirror Image) file. These .BMI files are listed to the right of the BASIC listing in the main menu. If you wish to suppress the creation of .BMI files, you should end your programs with either a QUIT command or SYSTEM command. These commands exit BASIC and return you to the main menu.

Exiting BASIC

There are 3 ways to exit BASIC. Press [CTRL] [F10] or type QUIT to return to the main menu. Press [CTRL] [F9] to exit BASIC and return to the previously run application program.

> **Note:** You may also exit BASIC with the SYSTEM command. This command suppresses the creation of .BMI files, however, and if you forget to save your program to disk or RAM, the computer deletes it.

GENERAL INFORMATION

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

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

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

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

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

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

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

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

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

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

Editing

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

For extensive editing of BASIC programs, you may save the program in ASCII format and then use the WORD utility. A BASIC program may be saved in ASCII by specifying the [A] option in the SAVE statement. For example:

```
SAVE "TEST",A
```

makes a copy of TEST in memory in ASCII format which may then be edited by WORD.

If you only need to edit a few lines, you may use the BASIC EDITOR. This editor is invoked by typing:

```
EDIT LINE NUMBER
```

in direct mode or when a BASIC program encounters the EDIT statement during execution.

After you invoke the EDIT command, the BASIC statement is printed and an edit cursor is positioned after the BASIC line number. The edit cursor is represented by a reverse video character. The editor cursor does not blink.

As you type characters, the new characters overwrite existing characters or are inserted depending on the mode you select.

In EDIT Mode, certain characters have special significance. These are as follows:

EDIT Control Characters

Name	KEY	Action
Cursor Left	[←]	Move the cursor to the preceding logical character on the line. If the cursor is at initial cursor position, the error tone is sounded.
Cursor Right	[→]	Move the cursor to the next logical character on the line. If the cursor is at the end of the line, the error tone is sounded.
Cursor Down	[↓]	If the line wraps to the next line (or lines), the cursor down key moves the cursor to the next line. The cursor cannot be moved to a location where there is no character.
Cursor Up	[↑]	Moves the cursor up to the next physical line of the logical BASIC line. (See Cursor Down).
Delete	[SHIFT] [DEL/BKSP]	The character beneath the cursor is deleted. If the line is empty, the error tone is sounded. If the cursor is on a carriage return character, the carriage return is deleted and this line and the following line are joined.

Backspace	DEL/BKSP	The character to the left of the cursor is deleted. The new line is then displayed. If the cursor is at the initial position, the character beneath the cursor is deleted.
Tab	TAB	The appropriate number of spaces are inserted to move character under the cursor to the next TAB position on the current line. An error tone is sounded if the farthest TAB position has already been reached.
Carriage Return	CTRL J	The physical line is broken at the current cursor position, but the logical line is preserved. Subsequent characters are moved to column 0 of the following line. The line scrolls if necessary. If the scroll forces any part of the line off of the screen, the error tone is sounded and the scroll doesn't occur.
ENTER	ENTER	The program line is returned to the program, and the editing session ends.
Break	CTRL C	This ends the editing session. The program line is unchanged.
Insert/ Overwrite Mode	CTRL R	Control R toggles between insert and overwrite modes. Pressing any of the cursor keys ends insert mode.
Escape	ESC	Deletes the current line from the screen and ends the editing session. The line remains unchanged in memory.

Text Characters

Any character or number	The character is inserted/overwritten at the cursor position and the new line is displayed. If the new line causes any part of the logical line to scroll off of the screen, the error tone is sounded. If the cursor is at the end of the line, the character is appended to the existing line. Line wrap occurs as necessary based on the current screen width.

Special Function Keys

Handheld BASIC assigns the following commonly used commands to the 10 function keys at the top of the Tandy 600 keyboard:

F1	LIST	F6	EDIT
F2	RUN←	F7	TRON←
F3	LOAD"	F8	TROFF←
F4	SAVE"	F9	FILES"
F5	CONT←	F10	KEY

You must press [ENTER] after pressing [F1], [F3], [F4], [F6], [F9], and [F10]. The [F2], [F5], [F7], and [F8] function keys contain a built-in [ENTER] command.

BASIC CONCEPTS

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

Elements of a Program

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

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

Here is a sample program line:

```
10 PRINT "one"
```

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

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

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

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

This program line has 3 statements. They are:

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

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

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

Data

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

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

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

Here are some sample strings:

"GARY"	"SHERRY ST"	"255 CENTRAL AVE"
"25 dollars"	"$250"	"16342"

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

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

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

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

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

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

1 3200 -2 500 -12345

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

Single precision numbers can be a maximum of 6 digits and may have a decimal point. Single precision numbers must be in the range 10^{-38} to 10^{+38}. Sample single precision numbers are:

10.001 -200034 123.4567

If a single precision number is more than 6 digits, BASIC displays the number in scientific notation, or exponential format, in the E form. For example:

1.74E 6.98E8 104E-7

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

Double precision numbers can include a maximum of 14 digits and may have a decimal point. Double precision numbers have the same range as single precision numbers. Sample double precision numbers are:

1010234567 -8.7777651010

If a double precision number is more than 14 digits, BASIC displays the number in scientific notation, or exponential format, in the D form. For example:

8.00100708D12 -6.7765499824D16

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

> **Note:** The Model 600 uses double precision as the default for numeric operations.

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

Hex	Decimal
&H76	118
&H02FF	767
&HFF	255

BASIC stores hexadecimal numbers as integers.

Octal numbers are the octal representation of decimal numbers. They contain 1 to 6 digits and are preceded by &O or &. Although only the & is required, we recommend that you use

&O for clarity in your programs. The octal numbers are 0, 1, 2, 3, 4, 5, 6, and 7. Here are some octal numbers and their decimal equivalents:

Octal	Decimal
&7	7
&O123	83
&O000456	302

BASIC stores octal numbers as integers.

Constants

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

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

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

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

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

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

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

Here are some examples of constants:

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

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

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

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

Variables

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

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

The following are examples of variable names:

A	A1	ADDRESS	ADDRESS.OLD
L	L2	LEN2	LENGTH

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

The following examples assign a value to a variable.

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

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

Declaring Numeric Constants and Variables

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

Numeric Constants

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

! declares a single precision number. For example, BASIC stores the number 12.345678901234! as a single precision number: 12.34568.

E declares the number a single precision exponential number. For example, BASIC stores the number 1.2E5 as a single precision number: 120000.

declares a double precision number. For example, BASIC stores the number 1.5# as a double precision number: 1.5. BASIC does not expand constants when declaring them double precision.

D declares the number a double precision exponential number. For example, BASIC stores the number 1.2D2 as a double precision number: 120.

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

Numeric Variables

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

- Append a symbol to the variable name:

  % declares an integer variable. BASIC stores the value of the variable as an integer. I%, FT%, and COUNTER% are samples of integer-declared variables.

  ! declares a single precision variable. BASIC stores the value of the variable as a single precision number. F!, NM!, and BALANCE! are samples of variables declared as single precision.

  # declares a double precision variable. BASIC stores the value of the variable as a double precision number. S#, AD#, and YTDTOTAL# are samples of variables declared as double precision.

  $ declares a string variable. The value of the variable must be enclosed in double quotes. A$, WRD$, and CITY$ are samples of variables declared as string variables.

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

- Use the following BASIC statements:

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

Chapter 10 describes these BASIC statements fully.

Numeric Precision Conversion

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

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

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

When converting *double* to *single precision*, BASIC rounds the number to 6 significant digits. For example:

A! = 1.2345678901234567	BASIC stores as 1.23457
A! = 1.3333333333333333	BASIC stores as 1.33333

Since the Tandy 600 uses the BCD (Binary Coded Decimal) system to perform mathematical operations, trailing zeroes are not added to double precision numbers less than 14 digits long. For example, even though BASIC stores A# = 1.5 as 1.5000000000000, it displays A# = 1.5 without the trailing zeroes.

Manipulating Data

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

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

An expression might look like this:

operand1	operator	operand2
6	+	2

A few operators allow only one operand, for example:

operator	operand
	5

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

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

BASIC has four types of operators:

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

Arithmetic Operators

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

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

^	Exponentiation. Calculates the power of a number. For example, 2^3 is 8 (2 to the power of 3 is the same as 2*2*2).
–	Negation or Unary Minus. Makes a number negative. For example, -10 is "negative ten."
*, /	Multiplication, Division. For example, 3*3 is 9, and 10/5 is 2.

\ Integer Division. BASIC rounds both operands to integers and truncates the result to an integer. Integer division is faster than standard division. For example, 10\4 is 2.

MOD Modulus Arithmetic. BASIC performs integer division as described above and returns the *remainder* as an integer value. For example, 10 MOD 3 results in 1.

+, − Addition, Subtraction. For example, 2+9 is 11, and 15-8 is 7.

String Operator

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

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

prints APRIL SHOWERS BRING MAY FLOWERS.

Relational Operators

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

The relational operators are, in order of precedence:

= Equal. Both operands are equal.

< Less Than. The first operand is less than or precedes the second operand.

> Greater Than. The first operand is greater than or follows the second operand.

>< or <> Inequality. The operands are not equal.

<= or =< Less Than or Equal To. The first operand is less than (precedes) or is equal to the second operand.

>= or => Greater Than or Equal To. The first operand is greater than (follows) or is equal to the second operand.

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

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

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

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

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

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

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

Consider these examples:

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

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

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

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

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

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

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

Logical Operators

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

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

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

Operator	Meaning of Operation	First Operand	Second Operand	Result
IMP	The result is true unless the first value is true and the second value is false.	1	1	1
		1	0	0
		0	1	1
		0	0	1

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

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

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

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

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

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

Hierarchy of Operators

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

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

Consider this expression:

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

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

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

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

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

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

Functions

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

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

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

Functions are described in Chapter 10.

ARRAYS

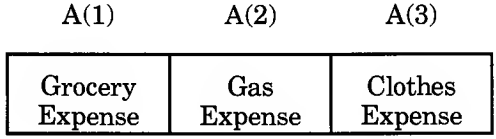
An array is a group of related data values stored consecutively in memory. The entire group of data values is referred to by one variable name. Each data value is called an *element* of the array. A *subscript* is an integer used to refer to each element of the array. For example, an array named A may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

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

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

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

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



Array A

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

This program creates a 1-dimensional array:

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

The DIM statement in Line 20 reserves space in memory for an array named *A* with 3 elements. As you enter the expenses, the grocery expense is stored in A(1), the gas expense in A(2) and the clothes expense in A(3).

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

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

Use RUN to see the results of this program.

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

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

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

A(row,column)

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

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

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

and change these lines:

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

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

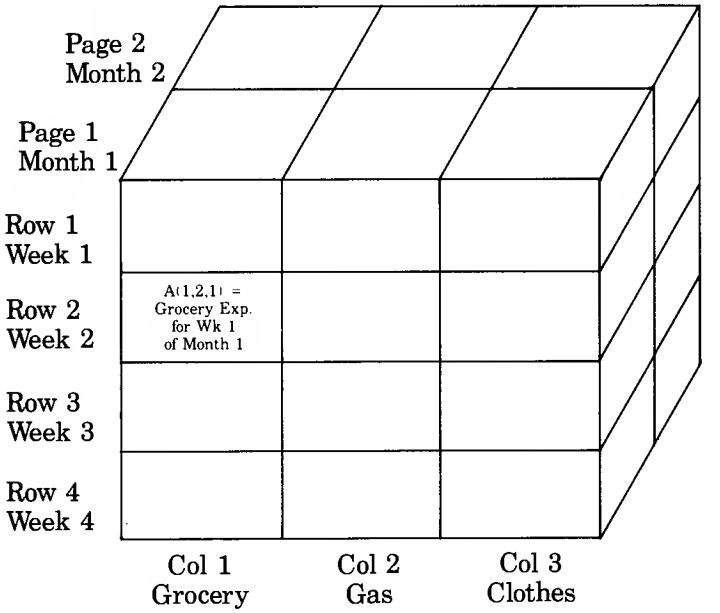
You can add yet another dimension, or subscript, to the array by adding these lines:

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

and changing these lines:

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

Run the program to see how it works.



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

A(page, row, column)

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

Types of Arrays

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

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

Defining Arrays

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

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

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

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

```
DIM C#(99)
```

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

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

Chapter 7

FILES

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

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

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

Sequential Access Files

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

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

The statements and functions used with sequential files are:

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

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

Creating a Sequential Access File

1. To create the file, open it in Output mode and assign it a buffer number in the range 1 to 15. For example:

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

   The OPEN statement opens a sequential output file named *list.dat* and gives Buffer 1 access to this file.

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

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

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

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

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

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

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

```
CLOSE 1
```

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

Sample Program

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

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

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

5. To access data in the file, reopen it in the Input mode. For example:

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

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

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

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

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

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

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

Sample Program

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

Updating a Sequential Access File

1. To add data to the file, first open it. For example:

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

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

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

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

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

Sample Program

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

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

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

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

Direct Access Files

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

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

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

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

The statements and functions used with direct access files are:

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

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

Creating a Direct Access File

1. To create the file, open it for random access in Random mode. For example:

```
OPEN "list.dat" AS 1 LEN=32
```

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

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

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

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

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

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

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

```
PUT 1, CODE%
```

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

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

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

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

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

Accessing a Direct Access File

1. Open the file in Random mode:

   ```
   OPEN "listing.dat" AS 1 LEN=32
   ```

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

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

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

For example:

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

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

```
GET 1, CODE%
```

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

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

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

The program may now access the data in the buffer.

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

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

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

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

If you enter a record number that is not part of *listing.dat*, your display shows:

```
$0.00
```

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

USING MACHINE LANGUAGE SUBROUTINES

Using DBCALLS.LIB

The Tandy 600's BASIC Interpreter, when used with the DBCALLS.LIB file supplied on the master diskette that came with your computer, gives you complete access to the same type of databases you create using FILE. You can add, delete, and move records; sort or modify them; and output the data in any form you desire. When you load the DBCALLS.LIB file into RAM and access it using a LIBRARY statement you essentially gain an additional 28 BASIC commands in the form of database CALLS. At the end of this chapter, there is a sample program that extracts records from a name-and-address database and prints them in mailing-label format using many of these new commands. During the explanation of the various DBCALLS we will use sample lines from this program as illustration. When an example of a particular call includes a line number, it comes from this sample program.

The general sequence of events is then as follows. First you open the FILE database. Next you determine the id numbers of the fields within the file which you will be using. Then you can open the record you desire, and read or write any selected fields within that record into string or numeric variables. After closing the record, you can repeat the process.

Before you can make use of the calls, you must load the DBCALLS.LIB file into RAM and access it within your BASIC program using the LIBRARY command:

```
LIBRARY "DBCALLS.LIB"
```

To close the library after using 1 or more of the subroutines, type:

```
LIBRARY CLOSE
```

Following are explanations of the 28 machine-language subroutines in the DBCALLS.LIB library.

DATABASE-ORIENTED CALLS

CALL DBCREATE ("filename". handle)

Creates a FILE-compatible database. The filename should include the .DAT extension if it will be used directly with FILE. A numeric variable, 'handle', is returned. This handle is used to identify the file in all later database calls.

EXAMPLE– CALL DBCREATE ("MAILLIST.DAT",H1)

CALL DBOPEN ("filename", access, handle)

Opens an existing database. The 'access' variable specifies whether the file is opened for reading, writing, or both, as follows:

Ø – Read only access
1 – Write only access
2 – Both read and write access

CALL DBOPEN returns a unique numeric "handle" that is used to specify the file in all subsequent database calls.

EXAMPLE– 20 READ A$: CALL DBOPEN (A$,Ø,H)

CALL DBCLOSE (handle)

This is the reverse of the above commands. It closes down the file specified by 'handle' which had earlier been opened using either the DBCREATE or DBOPEN call.

EXAMPLE– 500 PRINT "No More Records": CALL DBCLOSE (H)

CALL DBDELETE ("filename")

Deletes from memory the file with the name "filename."

EXAMPLE– CALL DBDELETE ("clients.dat")

CALL DBERROR (errorcode)

If an error occurs while using a database call, BASIC returns the generic "ILLEGAL DATABASE" error, whose code number is 91. When this happens, you should execute a CALL DBERROR. Into the numeric variable 'errorcode' a new code number is returned indicating what type of database error actually occurred. For a list of these database-specific error codes see Chapter 12.

```
EXAMPLE– 1000 IF ERR<>91 THEN PRINT ERR"in
         line"ERL:GOTO 500
         1010 CALL DBERROR(X)
         1015 IF X=66 THEN 500
         1020 PRINT "Database Call Error #"x"in
         line"ERL: GOTO 500
```

RECORD-ORIENTED CALLS

CALL RNUMF (handle, number-of-records)

This call returns the number of records in the database specified by 'handle' into the number-of-records numeric variable.

```
EXAMPLE– 25–CALL RNUMF(H,RT): PRINT "
         "RT"RECORDS IN IT."
```

CALL RCREATE (handle, [record-#])

Creates an empty record in the file specified by handle, at the position indicated by record-#. Valid Record-#'s range from 0 to 4095. Record-# can be omitted, in which case the record is created at the end of the file.

```
EXAMPLE– CALL RCREATE (H1,2): REM creates a
         record#2
```

CALL ROPEN (handle, record-#, [access])

Opens an existing record in preparation for reading, writing, or combined access as determined by the optional variable 'access.'

0 – read-only access
1 – write-only access
2 – combined read-write access

If this parameter is omitted, the type of access will default to the one chosen when the file was opened.

```
EXAMPLE– 200 FOR X=0 TO RT-1
         210 CLS: PRINT "RECORD #"X+1:
             PRINT
         215 CALL ROPEN (H,X)
```

CALL RCLOSE (handle, [record-#])

Close the record previously opened using RCREATE or ROPEN. Record-# is optional: when the call is completed it holds the number of the record which was closed.

```
EXAMPLE– 235 NEXT: CALL RCLOSE (H)
```

CALL RDELETE (handle, record-#)

Deletes the record with the specified record-#.

```
EXAMPLE– CALL RDELETE (DHANDLE, RNUMBER)
```

CALL RMOVE (handle, old-record-#, new-record-#)

Moves the record specified by old-record-# to the one specified by new-record-#.

```
EXAMPLE– CALL RMOVE (H,2,1):CALL RMOVE
         (H,3,2):CALL RMOVE (H,1,3)
```

CALL RFIND (handle, starting-rec-#, direction, query-key-array(), found-rec-#)

RFIND executes a search of the database, starting from 'starting-rec-#' and continuing in the specified direction (where 0=forward and 1=backward) until the next record is found which satisfies the conditions contained in the search-key-array. Search-key is a numeric array which is built using the database query calls described below. The matching record will have its record-# placed in the 'found-rec-#' variable.

```
EXAMPLE– 205 IF OF THEN CALL RFIND
         (H,X,0,Q(),X)
```

FIELD-ORIENTED CALLS

CALL FGID (handle, "fieldname", field-id)

Given a field name, the FGID call returns the id number of that field. The rest of the field-oriented database calls, including those which get information to and from a file, use this field-id to refer to the desired field. Case is not important in specifying the field name. ID#'s may not be in the same order as they appear when using FILE.

```
EXAMPLE– 30 READ FT: FOR X=1 TO FT
         40 READ F$(X),P(X): CALL
            FGID(H,F$(X),F(X))
         48 NEXT
         999 DATA 6,First.Name,1,Last.Name,1,
             Address,2,City,3,ST,3,Zip,3
```

CALL FGTYPE (handle, field-id, field-type)

This call returns in the numeric 'field-type' variable the kind of data which is in the field. Valid field types are: 1- Character/ String, 2- Numeric Double Precision, 3- Date.

```
EXAMPLE– 30 FOR X=1 TO FT
         42 CALL FGTYPE(H,F(X),T(X): REM
            place fieldtype in T
         48 NEXT
```

CALL FGNAME (handle, field-id, fieldname$)

Given the field-id number, the FGNAME call returns the field name into a string variable.

```
EXAMPLE– CALL FGNAME (H1, 2, f2$): 'put field w.
         id2 into f2$
```

CALL FCREATE (handle, "fieldname", fieldtype, field-id)

Creates a new field within the database specified by 'handle,' with the title of 'fieldname', and of the type specified by 'field-type.' (Valid fieldtypes are discussed as part of the FGTYPE call above.) The id of the new field is returned in the field-id numeric variable.

```
EXAMPLE– CALL FCREATE (FHANDLE, "Comments",
         1, FI)
```

CALL FDELETE (handle, "fieldname")

Deletes the field specified by "fieldname."

```
EXAMPLE– A$="Comments": CALL FDELETE
         (FHANDLE, A$)
```

CALL FPUT (handle, field-id, put-variable)

Puts data into the field specified by field-id. The put-variable which holds the data must match the type defined for that field. The field type can be determined by executing the FGTYPE call described above.

```
EXAMPLE– CALL FPUT (H1,F(1),"Juge"): CALL
         FGPUT (H1,F(2),"Ed")
```

CALL FGET (handle, field-id, get-variable)

Gets data from the specified field and puts it into get-variable. The file must be open to a specific record (see record oriented calls above.) The type of get-variable (numeric or string) must be the same as that defined for the field, which can be determined with the FGTYPE call.

If the data is a string, space must be allocated ahead of time to the variable which is to receive the field data. If the field contains more characters than does the get-variable string which will hold it, then data will be truncated to the exact length of the get-variable. If, however, the get-variable is longer than the string stored in the database field, the entire field will be read in and the get-variable length will be set to the length of the database data.

Database data is physically transferred to the receiving variable: therefore, if the get-variable is defined within the program itself, upon performing the FGET call, the actual program may be changed. To avoid this, it is highly recommended that such statements as:

```
A$ = "String for Receiving Data"
```

be changed to:

```
A$ = "String for Receiving Data" + " "
```

The concatentation of two strings forces the resultant string to be defined outside of program space, circumventing accidental modification of any program lines.

```
EXAMPLE–  6 ZZ$="      "+""
          220 FOR Y=1 TO
              FT
          225 D$(Y)=ZZ$
          230 IF T(Y)=2
              THEN CALL
              FGET (H,
              F(Y), D(Y))
              ELSE CALL
              FGET (H,
              F(Y), D$(Y))
          235 NEXT :
```

BASIC supports character strings and BCD (Binary Coded Decimal) numbers internally, and no special conversion is needed for these values.

Since this has no internal BASIC counterpart, date fields must be passed to the database library as an INTEGER array with the following assignments:

Array Element #

0	Year
1	Month
2	day
3	hour
4	minute
5	second
6	hundredths of second

The array passed must be passed by reference (i.e. "DATE()").

Date values, however, are somewhat different. The format of a date field in a database file is an 8 byte value with the following definition:

Byte 0-1: A 16 bit unsigned integer representing the year.

Byte 2: An 8 bit integer representing the month (1-12).

Byte 3: An 8 bit integer representing the day of the month (1-31).

Byte 4: An 8 bit integer representing the hour of the day in 24 hour format (0-23).

Byte 5: An 8 bit integer representing the minutes past the hour (0-59).

Byte 6: An 8 bit integer representing the seconds past the minute (0-59).

Byte 7: An 8 bit integer representing the hundredths of a second past the second (Ø-99).

CALL FRGET (handle, record-#, field-id, get-var)

Performs identically to FGET, except that the record specified in record -# is used rather than the current open record. This can eliminate the need to use the ROPEN and RCLOSE calls in some situations.

```
EXAMPLE– REC=1: FID=1: A$=" ";+" ":
          CALL FRGET (H1,REC,FID,A$)
```

CALL FRENAME (handle, old-name, new-name)

Renames the field specified by old-name to the name specified by new-name. Both must be string expressions. FRENAME does not affect field data — only field names.

DATA QUERYING

The following five calls are used to create a data query array. This is a numeric array which holds (in a coded form) the parameters which RFIND uses to discover matching records within a database. The array's size determines the possible complexity of the query.

CALL QSTART (handle, query-key-array() [,mode])

This call is used to begin the creation of a data query array. The query-key-array should be the name of the array that will be used to hold the completed query specification. The optional mode parameter is used to limit the size (and thus complexity) of the query. If it is not present or evaluates to zero, then the size of query is automatically computed and adjusted as the query is defined. Otherwise, a size may be predefined (it must be at least 7 bytes long) and the query defined by means other than the query calls.

```
EXAMPLE– 4 DIM Q(100)
         100 CALL QSTART (H, Q() )
```

CALL QRSTART (handle, query-key-array())

This call is used to start a new query record definition.

```
EXAMPLE– CALL QRSTART (H1, QU() )
```

CALL QFIELD (handle, query-key-array(), field-id, operation, modifier, match-data)

This call is used to define the actual query, specifying the fields(s) to search, the data to compare, and what type(s) of comparisons to make. It may be called more than once to create complex queries, so long as there is room in the data query array to hold the entire definition.

The query-key-buffer should be the name of the numeric array which is holding the data query specification (e.g.: Q().) The field-id specifies the field to use in the comparison. The operation parameter must evaluate to one of the following values:

0 meaning '=' (equal to)
1 meaning '<>' (not equal to)
2 meaning '<' (less than)
3 meaning '>' (greater than)
4 meaning '<=' (less than or equal to)
5 meaning '>=' (greater than or equal to)

The modifier parameter must be an integer numeric value. To DBCALLS.LIB it is bit-encoded; that is, each bit that makes up the value is used as a switch for a modification to the standard query. The legal values are as follows:

0 No special modifications to this databse query.

2 All "*" and "?" characters are to be interpreted as wild-card characters in string fields, and as a hex 'F' in date fields. The default treats them as part of the string and does not allow wild-card matches.

4 Indicates that case is not significant in string comparisons. The default assumes that case IS significant.

8 The match-data parameter in the call will specify the field-id of another field to be used in the comparison. The default assumes that the match-data parameter holds a value which the field is to be compared against.

These values may be added together to include more than one option in the modifier parameter. For instance, a modifier parameter of 6 indicates both wild-cards AND a case-insignificant match should be used.

The final parameter, 'match-data' holds a value that will be compared against the field specified by field-id. It could also hold

the name of a second field to be used in the comparison if the appropriate modifier option has been selected.

```
EXAMPLE– 120 INPUT "WHICH FIELD TO SEARCH
         UPON";SF
         150 INPUT "WHICH KIND OF
         MATCH";MT
         160 INPUT "WHAT DATA TO COMPARE
         AGAINST";MD$
         170 :CALL QFIELD (H, Q(), F(SF) , MT, 6,
         MD$)
```

CALL QEND (handle, query-key-array() [,actual-size])

This call should be made at the completion of a query definition. After a QEND call, no more query definition (QFIELD) calls are allowed unless a new QSTART call is made. The optional numeric parameter 'actual-size' returns the number of bytes that the completed query definition required.

```
EXAMPLE– 180 INPUT "Further Matching (Y or N)";
         A$
         185 IF A$="y" OR A$="y" THEN 110
         190 CALL QEND (H, Q()) : QF=1
```

CALL QREND (handle, query-key-array())

This call is used to close the current query record definition.

```
EXAMPLE– CALL QREND (H1, QU() )
```

SORTING

CALL DBSORT (handle, direction, field-id [, direction, field-id. . .])

This call sorts the database on the specified field-id(s) in a given direction. The sort is made in ascending or descending order, depending on the value of direction. (0 being ascending, and 1 being descending.) Multiple sorts can be performed, with the first one in the call taking highest priority. When additional data is added to the file, it is NOT automatically placed in the proper place. The sort should be executed again.

```
EXAMPLE- 60 X=0:INPUT"Which field# should the
            data be sorted by ";X
         70 IF X>0 THEN PRINT"Sorting by
            "F$(X);;CALL
            DBSORT(H,0,F(X)):PRINT"Sorted."
```

SAMPLE PROGRAM

The following BASIC program utilizes many of the DBCALLS.LIB calls to open an existing database and extract and print mailing labels from it. Information about the database is contained in the DATA statement on line 999, which can be customized to allow the program to work with different data files. The first item in the DATA statement is the name of the database. Next is a number indicating how many fields will be extracted from the database. Then each field name and the line on the mailing label that it is to be printed upon is listed. These must be in the order they are to be printed.

The program makes use of a number of variables, and the following is a list of some of the important variables and what they represent:

D() Array that holds any numeric data from the file.
D$() Array which holds all string data from the file.
F() Array of database field i.d. numbers.
F$() Array of database field names.
FT Total number of fields that program is interested in.
H Database handle; a unique numeric identifier for all db calls.
P() Array indicating label line on which to print each field.
Q() Array which holds critera for finding matching records.
QF Flag- whether to use all database records or do selective matching.
RT Total number of records in file.
T() Array of field data types (numeric, date, or string).
ZZ$ Blank string used as buffer when getting string data fields.

```
1 REM ***** TANDY 600  Database Mailing Labels
  Printer V01.00.00 *****
2 KEY OFF:CLS
4 DIM Q(100),F$(10),F(10),D$(10),D(10),T(10),
  P(10):REM set up all variables
6 ZZ$="                              "+"":REM allocate space
8 ON ERROR GOTO 1000
10 LIBRARY "DBCALLS.LIB": REM open databse calls
   library
14 REM
15 REM ****** Open file and check for fields *****
16 REM
```

```
20 READ A$: CALL DBOPEN(A$,0,H): REM read file name
   and open file
21 COLOR 1:PRINT "Opening Database "A$: COLOR 0
25 CALL RNUMF(H,RT):PRINT"   "RT"Records in it.":
   REM number-of-records
30 PRINT: READ FT: FOR X=1 TO FT
40 READ F$(X),P(X): CALL FGID(H,F$(X),F(X)): REM put
   each field.id in array F
42 CALL FGTYPE(H,F(X),T(X)): REM put each field.type
   in array T
44 PRINT "Field#"X,F$(X),
46 IF T(X)=2 THEN PRINT " -Numeric" ELSE PRINT " -
   Alpha"
48 NEXT
50 PRINT "ALL YOUR FIELDS ARE FOUND.":PRINT
54 REM
55 REM***** Sort the file on a field *****
56 REM
60 X=0:INPUT "Which field# should the data be sorted
   by";X
70 IF X=0 THEN PRINT "Not being sorted..."
75 IF X>0 THEN PRINT "Sorting by "F$(X);:CALL
   DBSORT(H,0,F(X)):PRINT ".Sorted."
80 INPUT "Select only certain records? (Y or N)";A$
90 IF A$="n" OR A$="N" THEN QF=0: GOTO 200
94 REM
95 REM ***** Create a query array to select specific
   records with *****
96 REM
100 CALL QSTART (H,Q()): REM start the definition
    process using array Q
110 CLS: FOR X=1 TO FT: PRINT: PRINT X"-- "F$(X),:
    NEXT: REM list the fields
120 INPUT "WHICH FIELD TO SEARCH UPON";SF
130 PRINT: PRINT " 0 -- Equal": PRINT " 1 -- Not
    Equal"
135 PRINT: 2 -- Less Than": PRINT " 3 -- Greater
    Than"
140 PRINT " 4 -- Less or Equal": PRINT " 5 --
    Greater or Equal",
150 INPUT "WHICH KIND OF MATCH";MT
160 PRINT: INPUT "WHAT DATA TO COMPARE
    AGAINST";MD$:MD=VAL(MD$)
170 IF T(SF)=2 THEN CALL
    QFIELD(H,Q(),F(SF),MT,6,MD):REM add a numeric
    compare
175 IF t(SF)<>2 THEN CALL
    QFIELD(H,Q(),F(SF),MT,6,MD$):REM...or string
    compare
180 INPUT " Further Matching? (Y or N)";A$
185 IF A$="y" OR A$="Y" THEN 110
190 CALL QEND(H,Q()): QF=1: REM finished creating
    the query array
194 REM
195 REM ***** Read the actual file *****
196 REM
200 FOR X=0 TO RT-1
```

```
205 IF QF THEN CALL RFIND(H,X,0,Q(),X): REM use Q
    array to find matching rec.#
215 CALL ROPEN(H,X): REM open the record
220 FOR Y=1 TO FT
225 D$(Y)=ZZ$: REM blank out some string space
230 IF T(Y)=2 THEN CALL FGET(H,F(Y),D(Y)) ELSE CALL
    FGET(H,F(Y),D$(Y)):'get it
235 NEXT: CALL RCLOSE(H): REM close record when done
    reading
237 REM
238 REM ***** Print a mailing label *****
239 REM
240 FOR Y=1 TO FT
245 IF T(Y)=2 THEN LPRINT D(Y)" "; ELSE LPRINT
    D$(Y)" ";
250 IF P(Y)<>P(Y+1) THEN LPRINT: REM start new line
    of the label
255 NEXT
260 FOR Z=P(Y-1) TO 5:LPRINT:NEXT: REM 3 spaces
    between labels
270 NEXT X
494 REM
495 REM ***** Close down the file and exit ****
496 REM
500 PRINT "No Further Records": CALL DBCLOSE(H): REM
    close file...
510 LIBRARY CLOSE: REM ...and the library
520 PRINT "PROGRAM RUN IS COMPLETE."
530 END
990 REM
995 REM ***** Program data and Error-handling *****
996 REM
999 DATA sample.DAT,6,First Name,1,Last Name,1,
    Address,2,City,3,ST,3,Zip,3
1000 IF ERR <>91 THEN PRINT "Error"ERR"in Line"ERL:
     GOTO 500
1010 CALL DBERROR(X): REM discover database-specific
     error code
1015 IF X=66 THEN 500: REM error 66 means 'no-
     records-match'
1020 PRINT "Database Call Error #"X"in line"ERL:
     GOTO 500
```

Chapter 9

INTRODUCTION TO BASIC KEYWORDS

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

Chapter 10 describes all of the Tandy 600's BASIC's keywords. This chapter explains the format used in Chapter 10. It also gives a quick summary of all of BASIC's keywords.

Format for Chapter 10

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

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

Some keywords are followed by defining words that explain how to use the command. An example is:

Trap used for event trapping

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

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

> **Important Note**: Tandy 600 Handheld BASIC requires that keywords be delimited by spaces. This means that you must leave a space between a keyword and any variables, constants, or other keywords. The only exceptions to this rule are characters that are shown as part of the syntax of the keyword.

For example, if you type:

```
SOUND1200,12 [ENTER]
```

BASIC returns a "Syntax error." You must leave a blank space between the word SOUND and the frequency and duration parameters.

Terms Used in Chapter 10

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

Statements

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

Statement	Description
BEEP	produces a sound from the computer speaker.
BREAK	enables, disables, or suspends restart trapping
CALL	calls an assembly-language subroutine.
CLEAR	sets all numeric variables to zero, all string variables to null, and closes all files.
CLOSE	closes access to a file.
CLS	clears the screen.
COLOR	toggles display background between normal and reverse video
CONT	continues program execution.
DATA	stores data in your program so that you can access it with a READ statement.
DEFDBL	defines variables as double precision.
DEF FN	defines a function according to your specifications.
DEFINT	defines variables as integers.
DEFSNG	defines variables as single precision.
DEFSTR	defines variables as strings.
DIM	defines the dimensions of an array.
EDIT	edits program lines.
END	ends a program.
ERL	returns the number of the line in which an error occurred.
ERR	returns an error code after an error.
ERROR	simulates the specified error.
FIELD	organizes a direct access buffer.
FILES	displays names of files in RAM or on a disk.
FOR/NEXT	establishes a program loop.
GET	gets a record from a direct access file.
GOSUB	transfers program control to a subroutine.
GOTO	transfers program control to the specified line.
IF/THEN/ELSE	evaluates an expression and performs an operation if conditions are met.
INPUT	accepts data from the keyboard.

Statement	Description
INPUT#	accepts data from a sequential access device or file.
INPUT$	accepts data from the keyboard or a sequential access file.
KEY	assigns or displays the current function-key soft values.
KEY/Trap	enables key-event trapping.
KILL	deletes a file.
LET	assigns a value to a variable. (The keyword LET may be omitted.)
LIBRARY	enables or disables libraries of machine language subroutines.
LINE/Graphics	draws a line on the display.
LINE INPUT	accepts an entire line from the keyboard.
LINE INPUT#	accepts an entire line from a sequential access file.
LIST	lists a program to the display or printer.
LLIST	prints a program on the printer.
LOAD	loads a program.
LOCATE	positions the cursor on the screen.
LPRINT	prints data at the printer.
LPRINT USING	prints data at the printer in a specified format.
LSET	moves data (and left-justifies it) to a field in a direct access file buffer.
MERGE	merges a program with a resident program.
MID$	replaces a portion of a string.
NAME	renames a file.
NEW	erases a program from RAM.
ON BREAK GOSUB	branches to a subroutine when the BREAK key is pressed.
ON ERROR GOTO	sets up an error-trapping routine.
ON/GOSUB	evaluates an expression and branches to a subroutine.
ON/GOTO	evaluates an expression and branches to another program line.
ON KEY() GOSUB	branches to a subroutine when a specific key is pressed.
ON RESTART GOSUB	branches to a subroutine when BASIC is restarted after QUIT or F9 or F10 halts operations.
ON TIMER() GOSUB	branches to a subroutine when timer equals the specified number.
OPEN	opens a file.

Statement	Description
PRESET/Graphics	draws a point in color at a specified position on the screen.
PRINT	lists data to the display.
PRINT USING	lists data to the display in a specific format.
PRINT#	writes data to a sequential access file.
PRINT# USING	writes data to a sequential access file using the specified format.
PSET/Graphics	draws a point on the screen at a specified position.
PUT	puts a record into a direct access file.
QUIT	suspends BASIC and activates another application.
RANDOMIZE	reseeds the random number generator.
READ	reads data stored in the DATA statement and assigns it to a variable.
REM	inserts a remark line in a program.
RENUM	renumbers a program.
RESET	closes all open files.
RESTART	enables, disables, or suspends restart trapping.
RESTORE	restores the DATA pointer.
RESUME	resumes program execution after an error-handling routine.
RETURN	returns from a subroutine to the calling program.
RSET	moves data (and right-justifies it) to a field in a direct access file buffer.
RUN	executes a program.
SAVE	saves a program.
SOUND	generates a specific tone for a specified length of time.
STOP	stops program execution.
SYSTEM	returns to main menu.
TIMER/Trap	controls timer event trapping.
TROFF	turns off the tracer.
TRON	turns on the tracer.
WRITE	prints data on the display.
WRITE#	writes data to a sequential file.

Functions

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

Function	Description
ABS	returns the absolute value of a number
ASC	returns the ASCII code of a character.
ATN	returns the arctangent of a number.
CDBL	converts a number to double precision.
CHR$	returns the character of an ASCII code.
CINT	converts a number to an integer.
COS	returns the cosine of a number.
CSNG	converts a number to single precision.
CSRLIN	returns the current row position of the cursor.
CVD	restores data from a direct access file to double precision.
CVI	restores data from a direct access file to integer.
CVS	restores data from a direct access file to single precision.
DATE$	sets the date or returns the current date.
EOF	checks for end-of-file.
EXP	returns the natural exponent of a number.
FIX	truncates to a whole number.
FRE	returns the number of bytes in memory not being used.
HEX$	converts a decimal value to a hexadecimal string.
INKEY$	returns the keyboard character.
INT	returns the integer value of a number.
LEFT$	returns the left portion of a string.
LEN	returns the length of the string.
LOC	returns the current file record number.
LOF	returns the total number of bytes in a file.
LOG	returns the natural logarithm of a number.
LPOS	returns the position of the print head in the printer buffer.
MID$	returns the midportion of a string.

Function	Description
MKD$	converts a double precision value to a string for writing it to a direct access file.
MKI$	converts an integer value to a string for writing it to a direct access file.
MKS$	converts a single precision number to a string for writing it to a direct access file.
OCT$	converts a decimal value to an octal string.
POINT	returns either the color of a point or current coordinates.
POS	returns the cursor column position on the display.
RIGHT$	returns the right portion of a string.
RND	returns a random number.
SGN	determines the sign of a number.
SIN	returns the sine of a number.
SPC	prints spaces to the display.
SQR	returns the square root of a number.
STR$	converts a number to a string.
TAB	positions the video cursor or the print head at a specified position.
TAN	returns the tangent of a number.
TIME$	sets the time or returns the current time.
VAL	returns the numeric value of a string.

BASIC KEYWORDS

ABS Function

ABS(*number*)

Returns the absolute value of *number*.

The absolute value of a number is the value without regard to its sign. Absolute values are always positive or zero.

Example

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

prints the absolute value of -66 which is 66.

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

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

Sample Program

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

ASC Function

ASC(*string*)

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

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

Example

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

prints 65, the ASCII code for A.

Sample Program

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

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

ATN Function

ATN(*number*)

Returns the arctangent of *number.*

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

BASIC always returns the result as a double precision number.

To convert this value to degrees, multiply the number returned by ATN by 180/pi.

Example

```
PRINT ATN(7)
```

prints the arctangent of 7 which is 1.4288992721907

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

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

BEEP Statement

BEEP

Sounds the speaker.

The BEEP statement sounds the ASCII bell character.

Example

```
20 IF X < 20 THEN BEEP
```

This executes a beep when X is less than 20.

BREAK ON/OFF/STOP — Statement

BREAK ON Enables restart trapping
BREAK OFF Disables restart trapping
BREAK STOP Suspends restart trapping

These statements are used in conjunction with the ON BREAK GOSUB statement. (See the description of that statement for more information.)

CALL Statement

CALL *routine name* [(*argument list*)]

Passes program control to an external subroutine located in one of the active library files.

routine name is the name of the library subroutine to which you wish to pass control. If more than one LIBRARY command has been issued, libraries are searched in reverse of the order in which they were opened.

argument list is an optional list of variables or constants, separated by commas, that are passed to the subroutine. (See Chapter 8 for a complete description of how to use this command.)

CDBL — Function

CDBL(*number*)

Converts X to a double precision number.

Example

```
10 LET PI = 22/7
20 PRINT CSNG (PI), CDBL (PI)
```

yields

```
3.14286       3.1428571428571
```

CHR$ Function

CHR$(*code*)

Returns a string whose one character is ASCII character *(code)*.

CHR$ is commonly used to send a special character to the screen or printer. For instance, a form feed (CHR$(12)) could be sent to clear the screen and return the cursor to the home position.

Example

```
PRINT CHR$(66)
```

yields

```
B
```

CINT Function

CINT(*number*)

Converts (*number*) to an integer by rounding the fractional portion.

If (*number*) is not in the range −32768 to 32767, an "Overflow" error occurs.

Example

```
PRINT CINT(45.67)
```

yields

```
46
```

CLEAR Statement

CLEAR

Sets all numeric variables to zero, all string variables to null, and closes all files.

The CLEAR statement performs the following actions:

Closes all files
Resets numeric variables and arrays to zero
Resets the stack and string space
Resets string variables and arrays to null
Resets all DEF FN and DEF SNG/DBL/STR statements

CLOSE Statement

CLOSE [[#]*file number*[,[#]*file number*. . .]]

Concludes I/O to a file. The CLOSE statement complements the OPEN statement.

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

The association of a particular file and a file number terminates after a CLOSE statement is executed. The file may then be reopened using the same or a different file number. Once a file is closed, that file's number may be used for any unopened file.

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

The SYSTEM, CLEAR, and END statements and the NEW and RESET commands always close all files automatically.

Example

```
CLOSE #1,#2
```

CLS Statement

CLS

Erases contents of entire current screen.

Example

```
10 CLS 'Clears the screen
```

COLOR Statement

COLOR [*parameter*]

Enables and disables reverse video mode.

If the optional numeric *parameter* is a non-zero value, reverse video is enabled. If *parameter* is zero or not present, reverse video is disabled.

CONT Statement

CONT

Resumes program execution.

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

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

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

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

Example

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

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

```
1, 2, 3 [ENTER]
```

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

```
PRINT L [ENTER]
```

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

```
C = 4
```

Now type:

```
CONT [ENTER]
```

and BASIC displays 144.

COS Function

COS(*number*)

Returns the cosine of *number*.

COS returns the cosine of the angle represented by *number*.

Number must be given in radians. If *number* is in degrees, use COS(number * pi/180) to convert *number* to radians.

BASIC always returns the result as a double precision number.

Examples

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

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

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

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

CSNG Function

CSNG(*number*)

To convert (*number*) to a single precision number.

```
10 A# = 975.3421115#
20 PRINT A#, CSNG(A#)
```

yields

```
975.3421115    975.342
```

CSRLIN Function

CSRLIN

Returns the current row position of the cursor.

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

Example

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

CVD, CVI, CVS — Function

CVD(*8- byte string*)
CVI(*2-byte string*)
CVS(*4-byte string*)

Converts string values to numeric values.

Numeric values that are read in from a direct access file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

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

Examples

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

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

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS (N$)
```

DATA Statement

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

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

This statement may contain as many *constants* (separated by commas) as can fit on a line (a maximum of 254 characters including the word DATA, commas, and spaces).

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

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

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

Note that expressions are not allowed in a DATA statement.

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

Examples

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

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

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

stores 4 numeric data items.

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

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

Sample Program

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

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

DATE$ Function

DATE$[= *string*]

Sets the date or retrieves the current date.

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

Setting the Date

You may use either a slash or a hyphen to separate the month, day, and year. You may use any of the following forms to set the current date:

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

The month (mm) may be any number Ø1-12.
The day (*dd*) may be any number Ø1-31.
The year (*yy* or *yyyy*) may be Ø1-99 or 1980-2099.

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

Retrieving the Date

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

mm-dd-yyyy

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

Examples

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

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

```
PRINT DATE$
```

prints the current system date.

```
CURDATE$ = DATE$
```

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

DEFDBL/INT/SNG/STR Statement

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

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

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

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

Examples

```
DEFDBL L-P
```

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

```
DEFSTR A
```

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

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

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

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

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

DEF FN Statement

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

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

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

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

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

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

Examples

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

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

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

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

DIM Statement

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

Sets aside storage for arrays with the dimensions you specify.

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

Dimension is 1 or more integer numbers separated by commas that define the dimensions of the array. The lowest element in a dimension is always zero.

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

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

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

For more information on arrays, see Chapter 6.

Theoretically, the maximum number of dimensions allowed in a DIM statement is 255. In reality, however, that number would be impossible, since the name and punctuation are also counted as spaces on the line, and the line itself has a limit of 255 characters.

If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array already dimensioned" error results. Therefore, it is good programming practice to put the required DIM statements at the beginning of a program, outside of any processing loops.

Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30   READ A(I)
40 NEXT I
 .
 .
 .
```

EDIT Statement

EDIT *line*

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

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

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

Examples

```
EDIT 100
```

enters Edit mode at Line 100.

```
EDIT .
```

enters Edit mode at current line.

END Statement

END

Ends program execution, closes all files, and returns to command level.

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

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

Sample Program

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

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

EOF Function

EOF(*file number*)

Detects the end of a file.

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

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

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

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

Example

```
10 OPEN "DATA" FOR OUTPUT AS 1
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
 .
 .
 .
```

ERL Statement

ERL

Returns the number of the line in which an error has occurred.

This function is primarily used inside an error-handling routine. If no error has occurred, ERL returns a Ø. If a statement entered at BASIC's prompt causes the error, ERL returns line number 65535 (the largest number that can be represented in 2 bytes).

Examples

```
PRINT ERL
```

prints the line number of the error.

```
E = ERL
```

stores the error's line number in variable E.

Sample Program

See ERROR.

ERR Statement

ERR

Returns the error number if an error has occurred.

ERR is only meaningful inside an error-handling routine accessed by ON ERROR GOTO.

See Chapter 12 for a list of error numbers and codes.

Example

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

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

Sample Program

See ERROR.

> **Note:** If you exit a BASIC program without using an error-trapping routine and an error occurs, you may use the PRINT ERR command to return the error number.

ERROR Statement

ERROR *code*

Simulates a specified error during program execution.

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

This statement is used mainly for testing an ON ERROR GOTO routine. When the computer encounters an ERROR statement, it proceeds as if the error corresponding to that code has occurred. (Refer to Chapter 12 for a listing of error codes and their meanings.)

Example

```
ERROR 1
```

causes a "NEXT without FOR" error (Code 1) when BASIC reaches this line.

Sample Program

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

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

EXP Function

EXP(*number*)

Returns the natural exponent of *number*, that is, *e* (base of natural logarithms) to the power of *number*.

Number must be less than or equal to 145.06286085

This functions is the inverse of the LOG function; therefore, *number* = EXP(LOG(*number*)).

BASIC always returns the result as a double precision number.

Example

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

prints the exponential value .13533528323661.

Sample Program

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

FIELD Statement

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

Allocates space for variables in a direct access file buffer.

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the direct access file buffer.

File number is the number under which you opened the file. *Field width* is the number of characters to be allocated to *string variable*.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that are executed remain in effect at the same time.

> **Note:** *Do not use a fielded variable name in an INPUT or LET statement.* Once a variable name is fielded, it points to the correct place in the direct access file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable no longer refers to the direct access file record buffer, but to the variables stored in string space.

Example 1

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

Allocates the first 20 bytes in the file buffer to the string variable N$, the next 10 bytes to ID$, and the next 40 to ADD$. FIELD does not place any data in the file buffer. (See also GET, LSET, and RSET Statements.)

Example 2

```
10 OPEN "PHONELST" AS 1 LEN=35
15 FIELD #1,2 AS RECNBR$,33 AS DUMMY$
20 FIELD #1,25 AS NAMES$,10 AS PHONENBR$
25 GET#1
30 TOTAL=CVI(RECNBR$)
35 FOR I=2 TO TOTAL
40   GET #1, I
45   PRINT NAMES$, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35-byte field is defined for the first record to keep track of the number of records in the file. In the next loop of statements (35-50), statement 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS
   A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0),16 AS A$(1),...,16 AS
A$(6),16 AS A$(7)
```

Example 4

```
10 DIM SIZE% (4%): REM ARRAY OF FIELD SIZES
20 FOR LOOP%=0 TO 4%
30   READ SIZE% (LOOP%)
40 NEXT LOOP%
50 DATA 9,10,12,21,41
 .
 .
 .
120 DIM A$(4%): REM ARRAY OF FIELDED VARIABLES
130 OFFSET%=0
140 FOR LOOP%=0 TO 4%
150 FIELD #1,OFFSET% AS OFFSET$,SIZE% (LOOP%) AS
    A$ (LOOP%)
160 OFFSET%=OFFSET%+SIZE% (LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4%) AS A$(4%)
```

FILES Statement

FILES [*filename*]

Prints the names of files residing in RAM or on disk.

If *filename* is omitted, all the files in RAM are listed. *filename* is a string formula which may contain question marks (?) or asterisks (*) used as wild cards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. The asterisk need not be used in the case where all the files on a drive are requested, e.g., FILES "A:".

FOR PAGE 99—

Examples

```
FILES
```

Shows all files in RAM.

```
FILES "*.BAS"
```

Shows all files with extension .BAS.

```
FILES "TEST?.BAS"
```

Shows all five-letter files whose names start with "TEST" and end with the .BAS extension.

FIX Function

FIX(*number*)

Returns the truncated integer of *number*.

Unlike CINT, FIX does not round the fractional portion of *number* when making it an integer. Instead, FIX simply strips the fractional portion from *number* so that the resultant value is a whole number. The result is the same precision as the argument (except for the fractional portion).

Unlike INT, FIX does not return the next lower number for a negative *number*.

FIX is the same as:

SGN(*number*)*INT(ABS(*number*)).

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

Examples

```
PRINT FIX (2.6)
```

prints 2.

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

prints -2.

FOR/NEXT Statement

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

NEXT [*variable*]

Establishes a program loop that allows a series of program statements to be executed a specified number of times.

Variable may be either integer, single precision, or double precision. Each FOR/NEXT loop must have a unique variable.

Increment is the number BASIC adds to the *initial value* each time the loop is executed. If you omit *increment*, BASIC increments by 1. If *increment* is a negative value, BASIC decreases the *initial value* each time through the loop. In this case, the *final value* must be less than the *initial value*.

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

> **Note:** BASIC skips the body of the loop if *initial value* is greater than *final value* when *increment* is positive or if *final value* is greater than *initial value* when *increment* is negative.

Sample Program

BASIC always sets the final value for the loop variable before setting the initial value. For example:

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

executes the loop 10 times, which prints:

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

Nested Loops

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

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

Sample Program

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

This program performs 3 outer loops and 2 inner loops within each of the outer loops. BASIC prints the following:

```
OUTER LOOP
    INNER LOOP
    INNER LOOP
OUTER LOOP
    INNER LOOP
    INNER LOOP
OUTER LOOP
    INNER LOOP
    INNER LOOP
```

By listing the counter variable, you can use the NEXT statement to close nested loops. (Be sure not to type the variables out of order.) For example, delete Line 920 and change 930 to:

```
930 NEXT J, I
```

If you omit the variables in nested loops, BASIC matches the most recent FOR statement.

FRE Function

FRE(*dummy argument*)

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

Dummy argument can be any string or numeric constant or variable. If you specify a numeric argument, BASIC returns the amount of memory available. If you specify a string argument, BASIC compresses the data before returning the amount of memory available. This frees unused memory that was once used for strings.

BASIC automatically compresses data if it runs out of workspace. This may take a few seconds.

Examples

```
PRINT FRE(44)
```

prints the amount of memory left.

GET Statement

GET [#]*file number*[,*record number*]

Reads a record from a random file into a random buffer.

File number is the number under which the file was OPENed. If *record number* is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215.

After a GET statement has been executed, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

Example

```
GET #1,75
```

GOSUB Statement

GOSUB *line*

Branches to the subroutine beginning at the specified line number.

Every subroutine must end with a RETURN. You can call a subroutine as many times as you want. When BASIC encounters a RETURN statement in the subroutine, it returns to the statement that follows the GOSUB.

GOSUB is similar to GOTO in that it may be preceded by a test statement.

Example

```
GOSUB 1000
```

branches to the subroutine at Line 1000.

Sample Program

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

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

GOTO Statement

GOTO *line*

Branches to the specified *line*.

When used alone, GOTO results in an unconditional branch. However, test statements, such as IF/THEN, may precede the GOTO to effect a conditional branch. Note that the GOTO is optional in IF/THEN statements. For example:

```
IF X=0 THEN 360 ELSE 200
```

BASIC branches to Line 360 if X equals 0. If not, BASIC branches to Line 200

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

Example

```
GOTO 100
```

BASIC transfers control to Line 100.

Sample Program

```
10 READ R
20 IF R = 13 THEN 80
30 PRINT "R=";R
40 A=3.14*R^2
50 PRINT "AREA =";A
60 GOTO 10
70 DATA 5,7,12, 13
80 END
```

Line 10 reads each of the data items in Line 70; Line 60 returns program control to Line 10. This enables BASIC to calculate the area for each of the data items until it reaches item 13.

HEX$ Function

HEX$(*number*)

Calculates the hexadecimal value of *number*.

HEX$ returns a string that represents the hexadecimal value of *number*. Since the value returned is like any other string, you cannot use it in a numeric expression. You cannot add hex strings; however, you can concatenate them.

Examples

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

prints the following strings 1E, 32, and 5A.

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

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

IF/THEN/ELSE — Statement

IF *expression* **THEN** *statement(s)*[**ELSE** *statement(s)*]

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

Expression is any numeric or string expression, usually making logical or relational comparisons.

Statement can be 1 or more valid BASIC statements. If there is more than 1 statement, they must be separated by colons. You can also specify a line number for BASIC to branch as a statement.

If *expression* is true, BASIC executes the THEN *statement*. If *expression* is false, BASIC executes the matching ELSE *statement* or the next program line.

You can also use IF/THEN to test the numeric value of a variable. If the variable contains a 0, the expression is true; otherwise, the expression is false.

Examples

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

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

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

tests the first expression. If it is true, BASIC prints $A < B$. Otherwise, BASIC jumps to the ELSE statement and prints $B <= A$.

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

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

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

branches to Line 210 if A$ is YES. If not, BASIC skips to the first ELSE, which introduces a new test. If A$ is NO, then

BASIC branches to Line 400. If A$ is any value besides NO or YES, BASIC branches to Line 370.

Sample Program

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

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

This program prints the relationship between the 2 numbers entered.

INKEY$ Function

INKEY$

Returns a keyboard character.

Returns a 1-character string from the keyboard without pressing [ENTER]. If there are characters already in the keyboard buffer, INKEY$ returns the first character from the buffer. If no key is pressed, BASIC returns a null string (length zero). INKEY$ does not echo the character to the display.

INKEY$ is invariably put inside some sort of loop. If not, program execution passes through the line containing INKEY$ before you can press a key.

The [BREAK] and [HOLD] keys are not passed to INKEY$. Also [ALT][CTRL][DELETE], which does a system reset, is not passed to INKEY$.

> **Note:** If your program contains an INKEY$ and you press a function key, BASIC returns 1 character of the key assignment at a time. For example, suppose this statement is executed:
>
> ```
> A$ = INKEY$
> ```
>
> Now suppose you press [F1], which initially has the value LIST. The first time the statement is executed A$ equals L, the second time A$ equals I, and so on. Keep this in mind when writing a BASIC routine to trap for a certain key. Your routine may not perform as expected if you accidently press a function key.

Example

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

causes the program to wait for you to press a key.

INPUT Statement

INPUT[;] [*"prompt"*;]*variable*[,*variable*,...]

Accepts data from the keyboard and inputs it into 1 or more variables. When BASIC encounters this statement, it stops execution and displays a question mark. This means that the program is waiting for you to type something.

Prompt is a string constant that BASIC displays before displaying the question mark prompt. *Prompt* must be enclosed in quotation marks, and follow the keyword INPUT. If, instead of a semicolon, you type a comma after *prompt*, BASIC suppresses the question mark when printing the prompt.

Variable may be 1 or more string or numeric variables to receive the input. If you specify more than 1 variable, separate them by commas.

If INPUT is immediately followed by a semicolon (;), BASIC does not echo the [ENTER] key when you press it as part of a response.

When typing multiple pieces of data on 1 line, separate the data items with a comma. The number of data items you supply must be the same as the number of variables you specify.

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

Examples

```
INPUT Y%
```

when BASIC reaches this line, you must type any number and press [ENTER] before the program can continue.

```
INPUT SENTENCE$
```

when BASIC reaches this line, you must type in a string. The string does not have to be enclosed in quotation marks unless it contains a comma, a colon, or a leading blank.

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

prints the prompt string on the screen, which helps the user enter the right kind of data.

Sample Program

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

INPUT# Statement

INPUT# *buffer, variable*[*,variable...*]

Accepts data from a sequential device or file and stores it in a program *variable.*

Buffer is the number assigned to the file when you opened it.

Variable is any string or numeric variable to contain the information.

The sequential file may be a disk file, a RAM file, or the keyboard device.

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

INPUT# does not care how you place the data in the file—whether you use a single PRINT# statement or 10 different PRINT# statements. INPUT# looks only for the position of the terminating characters and the end-of-file (EOF) marker.

When inputting data into a variable, BASIC ignores leading blanks. When the first nonblank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when BASIC encounters a terminating character or when a terminating condition occurs. The terminating characters vary, depending on whether BASIC is inputting to a numeric or a string variable:

> **Numeric:** BASIC ends input when it encounters a carriage return or a comma.
>
> **String:** BASIC ends input when it encounters a carriage return or a comma, unless the first character is a quotation mark("). If the first character is a question mark, BASIC ends input when it encounters a second quotation mark. Thus, a quoted string may not contain a quotation mark as a character.

Examples

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

sequentially inputs 2 numeric data items from the file opened to Buffer 1 and places them in A and B.

```
INPUT#4, A$, B$, C$
```

sequentially inputs 3 string data items from the file opened to Buffer 4 and places them in A$, B$, and C$.

INPUT$ Statement

INPUT$(*number* [,[#]*buffer*])

Accepts a string of characters from either the keyboard or a sequential access file.

Number is the number of characters to be input. It must be a value in the range 1 to 255.

Buffer is a buffer that accesses a sequential input file. If you include *buffer*, BASIC inputs the string from sequential access file. If you omit *buffer*, BASIC inputs the string from the keyboard. The number sign (#) is optional. It is provided for compatibility with other BASICs.

When inputting the string from the keyboard, BASIC waits until the user enters the number of characters specified by *number*. You do not need to press [ENTER] to signify end-of-line. The character(s) you type are not displayed on the screen. Any character, except [BREAK], is accepted for input.

When inputting from a sequential file, BASIC inputs the number of bytes specified by *number* from the file assigned to *buffer*.

Examples

```
A$ = INPUT$(5)
```

assigns a string of 5 keyboard characters to A$. Program execution halts until 5 characters are typed at the keyboard.

```
A$ = INPUT$(11,3)
```

assigns a string of 11 characters to A$. The characters are read from the file associated with Buffer 3.

Sample Program

In the program below, Line 100 opens a sequential input disk file (which we assume has been previously created). Line 200 retrieves a string of 70 characters from the file and stores them in T$. Line 300 closes the file.

```
100 OPEN "test.dat" FOR INPUT AS 2
200 T$ = INPUT$(70,2)
300 CLOSE
```

INT Function

INT(*number*)

Converts *number* to the largest integer that is less than or equal to *number*.

Number is not limited to the integer range - 32768 to 32767.

The result has the same precision as *number* (except for the fractional portion).

Unlike CINT, INT does not round positive numbers. It does, however, round negative numbers.

Examples

```
PRINT INT(79.89)
```

prints 79.

```
PRINT INT (-12.11)
```

prints -13.

KEY/Set/Display Statement

KEY *number,string*

KEY ON

KEY OFF

KEY LIST

KEY *number,string*

Assigns or displays function key values.

Number is an integer in the range 1 to 10 that indicates the function key being defined ([F1] – [F10]).

String is the string expression assigned to the key and may contain a maximum of 15 characters.

You can program the function keys on your computer to generate a specific string of characters. When you press the key, BASIC displays the string on the screen just as if you had typed every character. Initially, the function keys have these values:

F1	LIST	F7	TRON [ENTER]
F2	RUN [ENTER]	F8	TROFF [ENTER]
F3	LOAD"	F9	FILES"
F4	SAVE"	F10	KEY
F5	CONT[ENTER]		
F6	EDIT		

You also can use the KEY statement to redefine the other function keys so that BASIC displays the strings you use most often.

You can remove the string from a function key by assigning it a string length of zero (" "). For example:

```
KEY 1, ""
```

Key [F1] no longer has a string assigned to it.

KEY ON

KEY ON displays the function key assignment values at the bottom of the screen. The screen shows all 10 of the key assignments. However, the screen shows only the first 6 characters of the string. When you load BASIC, KEY ON is the initial default value.

KEY OFF

KEY OFF erases the soft key assignments from the bottom line. The assignments are still active, but the screen does not display them.

> **Note:** Typing KEY OFF greatly speeds display screen scrolling.

KEY LIST

KEY LIST displays all 15 characters of all 10 soft key assignments on the screen.

> **Note:** If your program contains an INKEY$ and you press a function key, BASIC returns 1 character of the key assignment at a time. For example, suppose this statement is executed:
>
> A$ = INKEY$
>
> Now suppose you press [F1], which initially has the value LIST. The first time the statement is executed A$ equals L, the second time A$ equals I, and so on. Keep this in mind when writing a BASIC routine to trap for a certain key. Your routine may not perform as expected if you accidentally press a function key.

KEY/Trap Statement

KEY(*number*) *action*

Turns on, turns off, or temporarily halts key trapping for a specified key.

Action may be any of the following:

ON	enables key trapping
OFF	disables key trapping
STOP	temporarily suspends key trapping

Number may be a number in the range 1 to 14, indicating the number of the key to trap. Function keys use their corresponding function key number (1-10). The cursor direction key trap numbers are:

[↑]	11
[←]	12
[→]	13
[↓]	14

The KEY() ON statement turns on key trapping for a specific key. BASIC checks after each program statement to see if the specified key has been pressed. If so, BASIC transfers program control to the line number specified in the ON KEY() GOSUB statement. For example:

```
KEY(3) ON
ON KEY(3) GOSUB 1000
```

BASIC turns on a trap for [F3]. BASIC continues to execute the other program statements, checking after each statement to see if [F3] has been pressed. When [F3] is pressed, BASIC branches to the subroutine beginning at Line 1000.

KEY() STOP temporarily halts trapping for the specified key. If the specified key is pressed, BASIC does not transfer program control to the ON KEY() GOSUB until you turn on key trapping again with a KEY() ON statement. BASIC remembers that the key was pressed and branches to the subroutine immediately after key trapping is turned on again.

KEY() OFF turns off key trapping. BASIC does not remember that the key has been pressed when key trapping is turned on again.

> **Note:** Key trapping only occurs while BASIC is running a program.

See ON KEY() GOSUB for more information on key trapping.

Sample Program

See ON KEY() GOSUB.

KILL Statement

KILL *filename*

Kills (deletes) files from disk or RAM.

You may delete any type of file. However, if the file is currently open, a "File already open" error occurs. You must close the file before deleting it.

KILL is used for all types of files: program files, random data files, and sequential data files. The filename may contain question marks (?) or asterisks (*) used as wildcards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at its position.

Example

```
200 KILL "DATA1?.DAT"
```

The position taken by the question mark matches any valid filename character. This command kills any file that has a six character name starting with "DATA1" and has the filename extension ".DAT". This includes "DATA10.DAT" and "DATA1Z.DAT".

LEFT$ Function

LEFT$(*string,number*)

Returns the specified number of characters from the left portion of *string*.

Number must be an integer in the range 1 to 255. If *number* is equal to or greater than the length of the *string*, BASIC returns the entire string.

Examples

```
PRINT LEFT$("BATTLESHIPS", 6)
```

prints BATTLE.

```
PRINT LEFT$("BIG FIERCE DOG", 20)
```

Since BIG FIERCE DOG is fewer than 20 characters, BASIC prints the whole phrase.

Sample Program

```
740 A$ = "TIMOTHY"
750 B$ = LEFT$(A$, 3)
760 PRINT B$; "--THAT'S SHORT FOR "; A$
```

When you run this program, BASIC prints:

```
TIM--THAT'S SHORT FOR TIMOTHY
```

Line 750 gets the 3 left characters of A$ and stores them in B$. Line 760 prints these 3 characters, a string, and the original contents of A$.

LEN Function

LEN(*string*)

Returns the number of characters in *string*. Blanks are counted.

Examples

```
X = LEN(SENTENCE$)
```

gets the length of SENTENCE$ and stores it in X.

```
PRINT LEN("CAMBRIDGE") + LEN("BERKELEY")
```

prints 17.

```
PRINT LEN("ORLANDO, FLORIDA")
```

prints 16.

LET Statement

LET *variable = expression*

Assigns the value of *expression* to *variable.*

Variable is a numeric or string variable.

Expression is a numeric or string constant or expression. A BASIC function can be substituted for *expression.*

BASIC does not require assignment statements to begin with LET, but you might want to use LET to be compatible with versions of BASIC that do require it.

Examples

```
LET A$ = "A ROSE IS A ROSE"
LET B1 = 1.23
LET X = X - Z1
LET X = SQR(B)
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant, expression, or function on the right side.

Sample Program

```
550 P = 1001: PRINT "P =" P
560 LET P = 2001: PRINT "NOW P ="P
```

LIBRARY Statement

LIBRARY *library name*
LIBRARY CLOSE

Enables/disables a library's files to be searched for a subroutine to execute.

Library name is a string expression that indicates the name of a library file used to search for subroutines specified in the CALL statement.

The CLOSE option allows the user to remove ALL libraries from the list of active libraries being searched on CALL statements. It is not possible to remove specific libraries from the list.

(See Chapter 8 for a more complete description of this command.)

LINE/Graphics Statement

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

Draws a line or a box on the video display.

The STEP option tells BASIC that the (*x,y*) coordinates are relative to the last point referenced. If you use STEP with the second set of coordinates, the coordinates are relative to the first set of coordinates.

(*x1,y1*) specifies the point at which to begin the line. *x1* is the horizontal coordinate, and *y1* is the vertical coordinate. If you omit (*x1,y1*) BASIC begins the line at the last point referenced on the screen.

(*x2,y2*) specifies the point at which to end the line. *x2* is the horizontal coordinate and *y2* is the vertical coordinate.

Color indicates the color of the line (black or white).

If you specify coordinates that are not in the range of the current viewport, BASIC displays only that portion of the line that is within the viewport.

With the **B** option, BASIC draws a box. The points that you specify are opposite corners.

If you specify both the **B** and **F** options, BASIC draws a box and fills the box in.

Examples

```
LINE -(319, 100)
```

draws a line from the last point referenced to point 319,100. This is the simplest form of the LINE statement. Note that when you omit the beginning points you must still include the hyphen.

```
LINE (0,0)-(319,100)
```

draws a diagonal line on the display.

```
LINE (0,100)-(319,100),1
```

draws a vertical line across the display in the background color.

```
LINE (0,0)-(100,100),,B
```

draws a box in the upper left corner of the display.

```
LINE (0,0)-(100,100),1,BF
```

draws a box on the display and fills it in.

Sample Programs

```
10 CLS
20 LINE -(RND*720,RND*128)
30 GO TO 20
```

Lines 10-30 create a loop that draws random lines on the video display.

```
40 FOR X=0 TO 720
50 LINE (X,0)-(X,120),X AND 1
60 NEXT
```

Lines 40-60 draw an alternating pattern, turning on and off the line.

```
10 CLS
20 LINE -(RND*639,RND*199),RND*2,BF
30 GO TO 20
```

This program draws a random filled boxes across the screen and displays alternate color patterns.

LINE INPUT Statement

LINE INPUT[;][*"prompt"*;] *string variable*

Accepts an entire line (a maximum of 254 characters) from the keyboard. LINE INPUT is a convenient way to input string data without accidental entry of delimiters (commas, quotation marks, etc.).

Prompt is a string constant enclosed in quotation marks that BASIC prints before waiting for input.

String variable is the variable to receive the input.

The only way to terminate the string input is to press [ENTER]. However, if LINE INPUT is immediately followed by a semicolon, pressing [ENTER] does not echo a carriage return to the display.

> **Note:** You must place a space between LINE and INPUT.

LINE INPUT is similar to INPUT, except:

- BASIC does not display a ? when waiting for input.
- Only 1 variable can be assigned at a time.
- Commas and quotation marks can be entered in the string input.
- Leading blanks are not ignored.

> **Note:** A LINE INPUT statement may be aborted by pressing [SHIFT] [BREAK]. BASIC then returns to the command level. If you are using the interpreter, type CONT to resume execution at the LINE INPUT.

Examples

```
LINE INPUT A$
```

waits for input to A$ without displaying a prompt.

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

displays the message and waits for input.

LINE INPUT# Statement

LINE INPUT#*buffer*, *variable*

Accepts an entire line of data from a sequential file to a string *variable*.

Buffer is the number assigned to the file when you opened it.

This statement is useful when you want to read an ASCII format BASIC program file as data or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT# reads everything from the first character up to:

- the end-of-file (control Z)
- the 255th data character
- a carriage return

Other characters encountered—quotation marks, commas, leading blanks—are included in the string.

> **Note:** You must place a space between LINE and INPUT#.

Example

If a ASCII format program file looks like this:

```
10 CLEAR 500
20 OPEN 1, "prog"
```

then the statement:

```
LINE INPUT#1, A$
```

can be used repeatedly to read each program line, one at a time.

LIST Statement

LIST *startline-endline* [,*device*]

Lists a program in memory to the display.

Startline specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program.

If you omit both *startline* and *endline*, BASIC lists the entire program.

Device may be either SCRN: (screen) or LPT1: (line printer 1). If you omit *device*, the lines are listed to the screen.

You can temporarily stop the listing by pressing [PAUSE]. Press any key again to continue the listing.

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

Examples

```
LIST
```

displays the entire program.

```
LIST 50-85, "SCRN:"
```

displays lines in the range 50 to 85 on the screen.

```
LIST .
```

displays the last program line that you have entered or edited and all higher numbered lines on the screen.

```
LIST -227
```

displays all lines up to and including 227 on the screen.

```
LIST 227- ,"LPT1:"
```

lists Line 227 and all higher numbered lines to the printer.

LLIST Statement

LLIST *startline-endline*

Lists program lines in memory to the printer.

Startline specifies the first line to be listed. If you omit *startline*, BASIC starts with the first line in your program.

Endline specifies the last line to be listed. If you omit *endline*, BASIC ends with the last line in your program.

If you omit both *startline* and *endline*, BASIC lists the entire program.

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

LLIST assumes an 80-character-wide printer.

Examples

```
LLIST
```

lists the entire program to the printer. To stop this process, press [PAUSE]. This causes a temporary halt in the computer's output to the printer. Press any key again to continue printing.

```
LLIST 68-90
```

prints lines in the range 68 to 90.

LOAD Statement

LOAD *filename* [,R]

Loads a BASIC program from disk into memory.

Filename is a standard file specification used to save the file to disk.

The R option tells BASIC to run the program. (LOAD with the R option is equivalent to the command RUN *filename*.) When you specify the R option, BASIC leaves all open files open and runs the program automatically. If you omit the R option, BASIC wipes out any resident BASIC program, clears all variables, and closes all open files.

> **Note:** You can press [SHIFT] [BREAK] at any time during LOAD, between files, or after a time-out period. BASIC exits the search and returns to BASIC's prompt. Previous memory contents remain unchanged.

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 occurs.

Example

```
LOAD "A:prog1.bas"
```

loads *prog1.bas* from Drive A, and then returns to the command mode.

LOC Function

LOC(*buffer*)

Returns the current record position within a file.

Buffer is the number assigned to the file when you opened it.

You use LOC to determine the current record position, that is, the number of the last record processed since you opened the file.

When used with direct access files, LOC returns the record number accessed by the last GET or PUT statement.

When used with sequential files, LOC returns the number of 128-byte blocks that have been read or written.

Example

```
IF LOC(1)>55 THEN END
```

Program execution ends, if the current record position is greater than 55.

Sample Program

```
1310 A$ = "WILLIAM WILSON"
1320 GET 1
1330 IF N$ = A$ THEN PRINT "FOUND IN RECORD"
LOC(1): CLOSE: END
1340 GOTO 1320
```

This is a **portion** of a direct access program. Elsewhere the file has been opened and fielded. N$ is a field variable. If N$ matches A$, the record number in which it was found is printed.

LOCATE Statement

LOCATE [*row*][,[*column*][,[*cursor*][]

Positions the cursor on the screen.

Row is a numeric expression in the range 1 to 24 that indicates the screen row where you want to position the cursor.

Column is a numeric expression that indicates the screen column where you want to position the cursor. It may be in the range 1 to 40 or 1 to 80, depending on the current screen width.

Cursor indicates whether the cursor is visible or invisible. Set *cursor* to 1 for a visible cursor and to 0 for an invisible cursor.

Examples

```
10 LOCATE 10,20,1
20 PRINT "PRINTING STARTED ON ROW 10, COLUMN 20"
```

positions the cursor on Row 10 in Column 20 and prints text.

```
10 LOCATE 11,1,0
20 PRINT "PRINTING STARTED IN ROW 11, COLUMN 1"
```

positions an invisible cursor in the first position of line 11. The cursor remains invisible until the LOCATE command is executed with the cursor option set.

LOF Function

LOF(*buffer*)

Returns the length of the file in bytes.

Buffer is the number assigned to the file when you opened it.

Example

```
Y = LOF(5)
```

assigns the length of the file in bytes to variable Y.

Sample Programs

During direct access to an existing file, you often need a way to know when you have read the last valid record. LOF provides a way:

```
1540 OPEN "unknown.txt" AS 1 LEN=128
1550 FIELD 1, 128 AS A$
1560 RCNUM% = 1     'START AT BEGINNING OF FILE
1570 RCSIZ% = 128   'SET RECORD SIZE
1580 IF RCNUM% * RCSIZ% > LOF(1) GOTO 1640
1590          'CHECK FOR END OF FILE
1600 GET 1, RCNUM% 'RECORD NUM. TO BE ACCESSED
1610 PRINT A$
1620 RCNUM% = RCNUM% + 1 'INCREMENT RECORD NUM
1630 GOTO 1580
1640 CLOSE
```

If you attempt to GET record numbers beyond the end-of-file, BASIC gives you an error.

These lines use LOF to determine where to start adding when you want to add to the end of a file:

```
1700 RCNUM% = (LOF(1) / RCSIZ%) + 1
1720          'HIGHEST EXISTING RECORD
1720 PUT 1, RCNUM%          'ADD NEXT RECORD
```

LOG Function

LOG(*number*)

Returns the natural logarithm of *number*.

Number must be greater than zero. LOG is the inverse of the EXP function.

BASIC always returns the result as a double precision number.

Examples

```
PRINT LOG(3.14159)
```

prints the value 1.1447290411851.

```
Z = 10 * LOG(P5/P1)
```

performs the indicated calculation and assigns the value to Z.

Sample Program

This program demonstrates the use of LOG. It 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 Function

LPOS(*number*)

Returns the logical position of the print head within the printer's buffer.

Number can be 0 or 1 to indicate LPT1:.

LPOS is only useful for checking the position of the print head after a LPRINT statement that is terminated by a semicolon to suppress the automatic carriage return. The statement containing LPOS is not executed until the LPRINT statement is finished printing.

LPOS does not necessarily give the physical position of the print head if the printed string contains the ASCII code for a carriage return. For example, if you are printing a string of 20 characters and the 10th character is the ASCII code for a carriage return, the printer advances to the next line after printing the ninth character before printing the remaining 10 characters. If the string is terminated by a semicolon to supress the automatic line feed, the physical location of the print head is at position 10, but LPOS returns a value of 21 because that is the logical location of the print head.

Example

You may want to use LPOS to determine whether there is enough room to continue printing more variables on the same line.

```
100 IF LPOS(X)>60 THEN LPRINT
```

If the printer has printed more than 60 characters, a carriage return is sent so that the printer skips to the next line.

LPRINT Statement

LPRINT [USING *format*;] *data*[,*data*,...]

Prints *data* on the printer.

LPRINT assumes a print width of 80 characters.

See PRINT and PRINT USING for more information on formatting the output.

Examples

```
LPRINT (A * 2)/3
```

prints the value of expression (A * 2)/3 on the printer.

```
LPRINT TAB(50) "TABBED 50"
```

moves the printer carriage to tab position 50 and prints TABBED 50. (Refer to the TAB function.)

```
LPRINT USING "#####.#"; 2.17
```

sends the formatted value ␣␣␣␣2.2 to the printer.

LSET Statement

LSET *field name = data*

Moves *data* to the direct access buffer and places it in *field name*, in preparation for a PUT statement.

Field name is a string variable defined in a FIELD statement.

You must have used FIELD to set up buffer fields before using LSET.

You must convert numeric values to string values before they are LSET. See MKI$, MKD$, MKS$.

You use LSET to left-justify the variable in the field. If the field is larger than the variable it is receiving, the field is filled with blanks on the right. If the variable is larger than the field, characters are truncated on the right. The complement command to LSET is RSET.

See also Chapter 7, "Files," and OPEN, CLOSE, FIELD, GET, PUT, and RSET.

Example

Suppose NM$ and AD$ have been defined as field names for a direct access file buffer. NM$ has a length of 18 characters; AD$ has a length of 25 characters. The statements:

```
LSET NM$ = "JIM CRICKET, JR."
LSET AD$ = "2000 EAST PECAN ST."
```

set the data in the buffer as follows:

```
JIM␣CRICKET,JR.␣␣␣   2000␣EAST␣PECAN␣ST.␣␣␣␣␣␣
```

Notice that filler blanks are placed to the right of the data strings in both cases. If we use RSET statements instead of LSET, the filler spaces are placed to the left. This is the only difference between LSET and RSET.

MERGE Statement

MERGE *filename*

Loads a BASIC program and merges it with the program currently in memory.

Filename is a standard file as described in Chapter 1. The filename is required. The file must be in ASCII format; that is, it must have been saved with the A option.

Program lines in *filename* are inserted into the resident program in sequential order. For example, suppose that 3 lines from *filename* are numbered 75, 85, and 90, and 3 lines from the resident program are numbered 70, 80, and 90. When you use MERGE on the 2 programs, this portion of the merged program is now numbered 70, 75, 80, 85, 90.

If line numbers on the new program coincide with line numbers in the resident program, the new program's lines replace the resident program's lines.

MERGE closes all files and clears all variables. Upon completion, BASIC returns its prompt.

Example

Suppose you have a BASIC program on disk, *prog2.bas* (saved in ASCII), that you want to merge with the program you have in memory:

```
MERGE "prog2.bas"
```

merges the 2 programs.

MID$ Statement

MID$(*oldstring*,*start*[,*length*]) = *newstring*

Replaces a portion of *oldstring* with *newstring*.

Oldstring is the variable name of the string you want to change.

Start is a number specifying the position of the first character you want to change.

Length is a number specifying the number of characters you want to replace. If *length* is omitted, all of *newstring* is used.

Newstring is the string to replace a portion of *oldstring*.

The length of the resultant string is always the same as the original string. If *newstring* is shorter than *length*, the entire replacement string is used.

Examples:

```
10 A$ = "LINCOLN"
20 MID$(A$,3,4) = "12345": PRINT A$
```

prints LI1234N.

Replace Line 20 with:

```
20 MID$(A$,5) = "01": PRINT A$
```

and BASIC prints LINC01N

MID$ Function

MID$(*string, start* [,*length*])

Returns a substring of a string.

Length is the number of characters in the substring. It must be in the range 1 to 255.

Start specifies the position in the string from which to get the substring.

If you omit *length* or if there are fewer than that number of characters to the right of *start* position, BASIC returns all characters to the right of the character at the *start* position including that character at *start.*

If *start* is greater than *number* of characters in *string*, BASIC returns a null string.

Examples

```
10 A$ = "WEATHERFORD"
20 PRINT MID$(A$, 3, 2)
```

prints AT.

```
F$ = MID$(A$, 3)
```

puts ATHERFORD into F$.

Sample Program

```
200 INPUT "AREA CODE AND NUMBER (NNN-NNN-NNNN)";
PH$
210 EX$ = MID$(PH$, 5, 3)
220 PRINT "NUMBER IS IN THE " EX$ " EXCHANGE."
```

The first 3 digits of a local phone number are sometimes called the exchange of the number. This program looks at a complete phone number (area code, exchange, last 4 digits) and picks out the exchange.

MKD$, MKI$, MKS$ Function

MKD$(*double precision expression*)
MKI$(*integer expression*)
MKS$(*single precision expression*)

Converts numeric values to string values.

Any numeric value that is placed in a direct file buffer with an LSET or RSET statement must be converted to a string.

These 3 functions are the inverse of CVD, CVI, and CVS. The byte values that make up the number are not changed; only 1 byte, the internal data-type specifier, is changed so that numeric data can be placed in a string variable.

MKD$ returns an 8-byte string; MKI$ returns a 2-byte string; and MKS$ returns a 4-byte string.

Example

```
LSET AVG$ = MKS$(0.123)
```

Sample Program

```
1350 OPEN "test.dat" AS 1 LEN=14
1360 FIELD 1, 2 AS I1$, 4 AS I2$, 8 AS I3$
1370 LSET I1$ = MKI$(3000)
1380 LSET I2$ = MKS$(3000.1)
1390 LSET I3$ = MKD$(3000.00001)
1400 PUT 1, 1
1410 CLOSE 1
```

For a program that retrieves the data from *test.dat*, see CVD/CVI/CVS.

NAME Statement

NAME *old filename* AS *new filename*

Renames *old filename* as *new filename.*

With this statement, the data in the file is left unchanged. *Old filename* must exist and *new filename* must not exist, otherwise, an error will result. Both files must also be on the same drive, or in RAM.

A file in RAM must be renamed in RAM, a file on Drive A must be renamed on Drive A. The error generated is "FA", if the above rules are not followed.

Old filename must be closed before the NAME command is executed. There also must be one free file handle.

Example

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

In this example, the file that was formerly named ACCTS is now named LEDGER.

NEW Statement

NEW

Deletes the program currently in memory and clears all variables. NEW also closes all open files, turns off the trace function and closes all open libraries.

Example

```
NEW
```

OCT$ Function

OCT$(*number*)

Returns the octal value of *number*.

OCT$ returns a string that represents the octal value of a decimal *number*. The value returned is like any other string—it cannot be used in a numeric expression.

Examples

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

prints the strings 36, 62, and 132.

```
Y$ = OCT$(X/84)
```

Y$ is a string representation of the integer quotient X/84 to base 8.

ON BREAK GOSUB Statement

ON BREAK GOSUB *line number*

Line number is the statement line number of the break event trap handler.

Branches to the specified subroutine when the break key is typed.

The program uses this trap to detect when the break key is typed. If you are not using break trapping, the break key stops any statement being executed and returns to command level. When you use break trapping, the program controls what happens (including ignoring the key) when the break key is typed.

The BREAK statement controls whether the trap is detected or not. The BREAK ON statement enables the GOSUB to occur.

If a BREAK OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a BREAK STOP statement has been executed, the GOSUB is not performed but will be performed as soon as a BREAK ON statement is executed.

When a BREAK GOSUB is executed, an automatic BREAK STOP is executed so that recursive traps cannot take place. The RETURN from the subroutine performs an automatic BREAK ON unless an explicit BREAK OFF is executed inside the subroutine.

(See also BREAK ON, BREAK OFF, BREAK STOP statements.)

ON ERROR GOTO Statement

ON ERROR GOTO *line number*

Enables error handling and specifies the first line of the error handling routine.

Once error handling is enabled, all errors detected, including direct mode errors (e.g., syntax errors), cause a jump to the specified error handling routine. If *line number* does not exist, an "Undefined line" error results.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error handling routine causes BASIC to stop and print the error message for the error that caused the trap.

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

Example

```
10 ON ERROR GOTO 1000
```

ON/GOSUB Statement

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

Looks at *n* and transfers program control to the subroutine indicated by the *n*th line listed.

For example, of *n* equals 1, BASIC branches to the first line listed; if *n* equals 2, BASIC branches to the second line listed.

Line is the subroutine line at which execution begins when BASIC makes the branch.

N must be a number in the range 0 to 255. If necessary, BASIC rounds *n* to an integer before evaluating it. If *n* is 0 or greater than the number of line numbers listed, BASIC continues with the next statement. If *n* is negative or is greater than 255, an "Illegal function call" error occurs.

Use the RETURN statement to exit the subroutine.

Example

```
10 ON Y GOSUB 1000, 2000, 3000
```

If Y equals 1, BASIC branches to a subroutine, beginning at Line 1000. If Y equals 2, BASIC branches to a subroutine, beginning at Line 2000. If Y equals 3, BASIC branches to a subroutine, beginning at Line 3000.

If Y is outside the range 1 to 3, BASIC either continues with the next statement or generates an "Illegal function call," as mentioned earlier.

Sample Program

```
430 INPUT "CHOOSE 1, 2, OR 3" ; I
440 ON I GOSUB 500, 600, 700
450 END
500 PRINT "SUBROUTINE #1": RETURN
600 PRINT "SUBROUTINE #2": RETURN
700 PRINT "SUBROUTINE #3": RETURN
```

ON/GOTO Statement

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

Looks at *n* and transfers program control to the *n*th line listed.

For example, if *n* equals 1, BASIC branches to the first line listed; if *n* equals 2, BASIC branches to the second line listed.

N must be in the range 1 to 255. If necessary, BASIC rounds *n* to an integer before evaluating it. If *n* is 0 or is greater than the number of line numbers listed, BASIC continues with the next statement. If *n* is negative or is greater than 255, an "Illegal function call" error occurs.

Example

```
10 ON MI GOTO 150, 160, 170, 150, 180
```

tells BASIC to evaluate MI. If MI equals 1, BASIC branches to Line 150; if MI equals 2, BASIC branches to Line 160; and so on. If MI is outside of the range 1 to 5, BASIC either continues with the next statement or generates an Illegal function call, as mentioned earlier.

Sample Program

```
5 REM <CAPS> MUST BE ON
10 INPUT "ENTER A,B, or C,";L$
20 L=ASC (L$)
30 ON L-64 GOTO 50, 60, 70
40 PRINT "TRY AGAIN":GOTO 10
50 PRINT "YOU TYPED 'A'":END
60 PRINT "YOU TYPED 'B'":END
70 PRINT "YOU TYPED 'C'":END
```

ON KEY GOSUB Statement

ON KEY(*number*) GOSUB *line number*

Specify the first line number of a subroutine to be executed when a specified key is pressed.

Number is the number of a function key, direction key, or user-defined key.

Line number is the number of the first line of a subroutine that is executed when the specified function or cursor direction key is pressed.

A *line number* of zero disables the event trap.

The ON KEY statement is executed only if a KEY ON statement has been executed to enable event trapping. If key trapping is enabled, and if the *line number* in the ON KEY statement is not zero, BASIC checks to see if the specified function, user-defined or cursor direction key has been pressed. If the key was pressed, the program branches to a subroutine specified by the GOSUB statement.

If a KEY OFF statement was executed for the specified key, the GOSUB is not executed.

If a KEY STOP statement was executed for the specified key, the GOSUB is not performed, but will be performed as soon as a KEY ON statement is executed.

When an event trap occurs and a GOSUB is executed, an automatic KEY STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine automatically performs a KEY ON statement unless an explicit KEY OFF was executed inside the subroutine.

The RETURN *line number* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. You should use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping does not take place when BASIC is not executing a program, and event trapping is automatically disabled when an error trap occurs.

The following rules apply to keys trapped by BASIC:

1. The line printer echo toggle key is processed first. Defining this key as a user-defined key trap will not prevent characters from being echoed to the line printer if depressed.

2. Function keys and the cursor direction keys are examined next. Defining a function key or cursor direction key as a user-defined key trap has no effect because they are already defined.

3. Any key that is trapped is not passed on. That is, the key is not read by BASIC. **This applies to all keys, including Break or system reset (warm boot)! This powerful feature allows you to prevent BASIC application users from accidentally BREAKing out of a program or rebooting the machine.**

   Note: When a key is trapped, the occurrence of the key is not retained in memory. Therefore, you cannot use the INPUT or INKEY$ statements to find out which key caused the trap. If you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

Example

The following program overrides the normal function associated with function key 4, and replaces it with FILES "A:", which is printed whenever that key is pressed. The value may be reassigned, and it resumes its standard function when the machine is rebooted.

```
10 KEY 4,"FILES "+CHR$(34)+"A:"+CHR$(34)
   'assigns softkey 4
20 KEY(4) ON 'enables event trapping
.
.
70 ON KEY(4) GOSUB 200
.
.
.
key 4 pressed
.
.
.
200'Subroutine for displaying disk files
```

ON RESTART GOSUB Statement

ON RESTART GOSUB *line number*

Branches to the specified subroutine when Basic is restarted after a suspend operation (QUIT, etc.).

Line number is the statement line number of the restart event trap handler.

This trap lets the program detect when it has been suspended. Since BASIC does not save the contents of the screen after a suspend operations command, this trap may be used to redraw the screen when BASIC is restarted.

The RESTART statement controls whether the trap is detected or not.

If a RESTART OFF statement was executed first, the GOSUB is not performed and is not remembered.

If a RESTART STOP statement was executed, the GOSUB is not performed but will be performed as soon as a RESTART ON statement is executed.

When a RESTART GOSUB is executed, an automatic RESTART STOP is executed so that recursive traps cannot take place. The RETURN from the subroutine performs an automatic RESTART ON unless an explicit RESTART OFF is executed inside the subroutine.

ON TIMER() GOSUB Statement

ON TIMER(*number*) GOSUB *line*

Transfers program control to a subroutine when the specified period of time has elapsed.

Number indicates the number of seconds. *Number* may be a value in the range 1 to 86400 (86400 seconds = 24 hours).

Line is the first line number in the subroutine to execute when the specified time has passed. Use RETURN to exit the subroutine.

BASIC executes the ON TIMER() GOSUB statement only if a TIMER ON statement has been executed previously to enable time event trapping.

If a TIMER STOP statement has been issued to halt time event trapping temporarily, BASIC executes the subroutine immediately after the next TIMER ON statement.

When you execute the ON TIMER() GOSUB statement, BASIC immediately issues a TIMER STOP to prevent recursive traps. When BASIC executes the RETURN from the subroutine, it automatically executes another TIMER ON statement to enable trapping again, unless the subroutine executes a TIMER OFF statement.

Example

```
10 TIMER ON
20 ON TIMER (60) GOSUB 1000
30 REM
.
.
.
500 END
.
.
.
1000 REM PROCESSING ROUTINE
.
.
.
1100 RETURN 30
```

Line 10 turns on timer trapping. After each statement is executed, BASIC checks to see if the specified time has elapsed. If it has, BASIC immediately executes the subroutine at Line 1000.

OPEN Statement

OPEN[*device:*]*filename* (**FOR** *mode* **AS** [#]*buffer* [LEN=*record length*]

Establishes input and output to a file or device.

filename is an optional device specification followed by a filename.

Device is a character device.

Mode is one of the following expressions:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode
APPEND	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A PRINT# or WRITE# statement then extends (appends) the file.

If *mode* is omitted, the default random access mode is assumed. Random, however, cannot be expressed explicitly as the file mode.

File number is an integer between 1 and 255. The number is then associated with the file for as long as it is OPEN and is also used when accessing the file in other disk I/O statements.

Record length is an integer that, if included, sets the record length for random files. The default length for records is 128 bytes.

Files

A file must be opened before any I/O operation can be performed on that file. OPEN allocates a buffer for file or device I/O and determines the access mode to be used with the buffer.

The LEN= option is ignored if the file being opened is a sequential file.

Devices

BASIC devices are:

KYBD: LPT1: SCRN:

The BASIC file I/O system allows the user to take advantage of user installed devices.

Character devices can be opened and used like disk files. However, characters are not buffered by BASIC as they are for disk files. The record length is set to one.

> **Note:** A file can be opened for sequential input or direct access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Examples

```
10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

The following line opens the printer for output:

```
100 OPEN "LPT1:" FOR OUTPUT AS #1
```

POINT/Graphics Function

POINT (*x,y*)
POINT (*action*)

Returns the color number of a point on the screen or returns the current graphic/cursor coordinates.

(*x,y*) specify the coordinates of the point. *x* is the horizontal point, and *y* is the vertical point. The *x* and *y* coordinates must be absolute values. If you specify a point that is out of range, BASIC returns a -1.

Action is one of the following:

0 returns the current physical x-coordinate (horizontal).

1 returns the current physical y-coordinate (vertical).

Examples

```
20 LET C=0
30 PSET (10,10)
40 IF POINT(10,10)=C THEN PRINT "This point is
   color ";C

10 IF POINT (i,i)<>0 THEN PRESET (i,i)
   ELSE PSET (i,i)
   'invert current state of a point
20 PSET (i,i),1-POINT(i,i) 'another way to
   invert a point.
```

POS Function

POS(*number*)

Returns the current column position of the cursor.

Number is a dummy argument.

POS returns a number in the range 1 to 80, indicating the current cursor-column position on the display.

Example

```
PRINT TAB(40) POS(0)
```

prints 40. The PRINT TAB statement moves the cursor to Position 40; therefore, POS(0) returns the value 40. (However, because a blank is inserted before the "4" to accommodate the sign, the "4" is actually at Position 41.)

Sample Program

```
150 CLS
160 A$ = INKEY$
170 IF A$ = "" THEN 160
180 IF POS(X) > 70 THEN IF A$ = CHR$(32) THEN A$
= CHR$(13)
200 LPRINT A$;
210 GOTO 160
```

This program lets you use your printer as a typewriter (except that you cannot correct mistakes). Your computer keyboard is the typewriter keyboard. Everything you type is printed on your printer. The program also makes sure that no word is divided between two lines.

PRINT Statement

PRINT *data*[,*data*,...]

Prints numeric or string *data* on the display. You can substitute a question mark (?) in place of the word PRINT.

Data is any numeric or string constant or variable. If you omit *data*, BASIC prints a blank line. If you specify more that 1 data item in the statement, separate them by commas, semicolons, or spaces.

If you use commas, the cursor automatically advances to the next tab position before printing the next item. (BASIC divides each line into print zones containing 14 positions each, at columns 14, 28, 42, 56, and 70.)

If you use semicolons or spaces to separate the data items, PRINT prints the items without any spaces between them. BASIC begins the next PRINT item where the last one stopped.

If no trailing punctuation is at the end of the PRINT statement, the cursor drops to the beginning of the next line.

If BASIC tries to print a string longer than it can fit on the current line, it moves to the next line and prints the string.

Single precision numbers with 7 or fewer digits that can be accurately represented are printed in regular format rather than exponential format. For example, 1E-7 is printed as .0000001; 1E-8 is printed as 1E-08.

Double precision numbers with 16 or fewer digits that can be accurately represented are printed in regular format rather than exponential format. For example, 1D-15 is printed as .000000000000001; 1D-16 is printed as 1D-16.

BASIC prints all numbers with a trailing blank and prints positive numbers with a leading blank. Negative numbers are preceded by a minus sign.

String constants must be enclosed in quotation marks.

Examples

```
PRINT "DO"; "NOT"; "LEAVE"; "SPACES"; "BETWEEN";
      "THESE"; "WORDS"
```

displays 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"
```

Because F and N are positive numbers, PRINT inserts a space before and after them; therefore, your display should look similar to this (depending on your input):

```
IN THE YEAR 2004 YOU WILL BE 46 YEARS OLD
```

If we had separated each expression in Line 100 by a comma:

```
100 PRINT "IN THE YEAR",F,"YOU WILL BE",N,"YEARS
OLD"
```

BASIC would move to the next tab position after printing each data item.

PRINT USING Statement

PRINT USING *format*; *data*[,*data*,...]

Prints data using a format you specified. This statement is especially useful for printing report headings, accounting reports, checks, or any other documents that require a specific format.

Format consists of 1 or more field specifier(s), or any alphanumeric character. *Format* must be enclosed in quotation marks.

Data may be string and/or numeric value(s). If you specify more than 1 data item in the statement, use the same separators as described in PRINT.

With PRINT USING, you may use certain characters called field specifiers, to format the field. You may use more than 1 field specifier, except as indicated.

Specifiers for String Fields:

! prints the first character in the string only.

```
PRINT USING "!"; "PERSONNEL"
```

BASIC prints P.

\spaces\ prints 2 + *n* characters from the string (n is the number of spaces between the slashes). If you type the backslashes without any spaces, BASIC prints 2 characters; with one space, BASIC prints 3 characters, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified and padded with spaces on the right.

```
PRINT USING "\  \"; "PERSONNEL"
```

BASIC prints PERSO

& prints the string without modifications.

```
10 A$="TAKE":B$="RACE"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
```

When this program is run, BASIC prints TRACE.

Specifiers for Numeric Fields:

\# prints the same number of digit positions as number signs (#). Numbers are rounded as necessary.

You may insert a decimal point at any position. BASIC always prints the digits preceding the decimal point. If there is no number, BASIC prints a zero.

If the number to be printed has fewer digits than positions specified, the number is right-justified (preceded by spaces). If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed in front of the number.

```
PRINT USING "##.##";111.22
PRINT USING "##.##";.75
PRINT USING "###.##";876.567
```

BASIC prints %111.22, 0.75 and 876.57, respectively.

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

\+ prints the sign of the number. The plus sign may be typed at the beginning or at the end of the format string.

```
PRINT USING "+##.## "; -98.45,3.50,22.22,-.9
```

BASIC prints: -98.45 +3.50 +22.22 -0.90

```
PRINT USING "##.##+ "; -98.45,3.50,22.22,-.9
```

BASIC prints: 98.45- 3.50+ 22.22+ 0.90-

(Note the use of spaces at the end of a format string to separate printed values.)

— prints a negative sign *after* negative numbers (and a space after positive numbers).

```
PRINT USING "###.#-"; -768.660
```

BASIC prints 768.7-.

** fills leading spaces with asterisks. The 2 asterisks also establish 2 more positions in the field.

```
PRINT USING "******"; 44.0
```

BASIC prints ****44

$$ prints a dollar sign immediately before the number. This specifies 2 more digit positions, one of which is the dollar sign. You may not use exponent format with $$.

```
PRINT USING "$$###.##"; 112.7890
```

BASIC prints $112.79

**$ fills leading spaces with asterisks and prints a dollar sign immediately before the number.

```
PRINT USING "**$###.##"; 8.333
```

BASIC prints ***$8.33

, prints a comma before every third digit to the left of the decimal point. The comma establishes another digit position.

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

BASIC prints 1,234.50

^^^^ prints in exponential format. The 4 exponent signs are placed after the digit position characters. You may specify any decimal point position. You may not use $$ with exponent format.

```
PRINT USING ".####^^^^"; 888888
```

BASIC prints .8889E+06

_ Prints next character as a literal character.

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

BASIC prints !12.34!

Sample Program

```
420 CLS: A$ = "**$##,######.## DOLLARS"
430 INPUT "WHAT IS YOUR FIRST NAME"; F$
440 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
450 INPUT "WHAT IS YOUR LAST NAME"; L$
460 INPUT "ENTER AMOUNT PAYABLE"; P
470 CLS : PRINT "PAY TO THE ORDER OF ";
480 PRINT USING "!! !! "; F$; "."; M$; ".";
490 PRINT L$
500 PRINT :PRINT USING A$; P
```

In Line 480, each ! picks up the first character of one of the following strings (F$, ".", M$, and "." again). Notice the 2 spaces in "!␣!␣". These 2 spaces insert the appropriate spaces after the initials of the name (see below). Also notice the use of the variables A$ for format and P for item list in Line 500. Any serious use of the PRINT USING statement would probably require the use of variables rather than constants, at least for data items. (We have used constants in our examples for the sake of better illustration.)

When the program above is run, the display shows:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.6
PAY TO THE ORDER OF J. P. JONES

*****$12,345.60 DOLLARS
```

PRINT# Statement

PRINT# *buffer*,[USING *format*] *data*[,*data*,...]

Writes data items to a sequential file.

Buffer is the number assigned to the file when you opened it.

When you first open a file for sequential output, BASIC sets a pointer to the beginning of the file—that is where PRINT# starts writing the data items. At the end of each PRINT# operation, the pointer advances so that data items are written in sequence.

A PRINT# statement creates a file image similar to the image a PRINT to the display creates on the screen. For this reason, be sure to delimit the data so that it will be input correctly from the file.

PRINT# does not compress the data before writing it to the file. It writes an ASCII-coded image of the data.

When you include the USING option, data is written to the file in the format you specify. See PRINT USING.

Examples

```
If A = 123.45
PRINT# 1,A
```

writes this 9-byte character sequence to the file as:

```
ø123.45ø carriage return
```

The punctuation in the PRINT list is very important. Unquoted commas and semicolons have the same effect as they do in regular PRINT statements to the display. For example:

```
A = 2300
B = 1.303
PRINT# 1, A,B
```

writes the data to the file:

```
ø 2300 øøøøøøøøøø 1.303ø carriage return
```

The comma tells BASIC to tab between A and B, which creates 10 extra spaces in the file. Generally you do not want to use up storage space this way, so you use semicolons instead of commas.

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

This time BASIC writes the data as:

```
123.45,1.303
```

An INPUT# statement reads this as 2 separate fields.

If string variables contain commas, semicolons, or leading blanks, enclose them in quotation marks. For example:

```
A$ = CAMERA, AUTOMATIC
B$ = 102382
PRINT# 1, A$; B$
```

writes the data as:

```
CAMERA␣␣␣␣␣␣␣␣␣␣AUTOMATIC102382
```

An INPUT# statement reads this as 2 separate fields

```
A$ = CAMERA
B$ = AUTOMATIC102382
```

To separate these 2 strings properly in the file, write quotation marks using the hexadecimal representation CHR$(34). For example:

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

BASIC writes the following image to the file:

```
"CAMERA,AUTOMATIC"102382"
```

The statement INPUT# 1, A$, B$ reads "CAMERA,AUTOMATIC" into A$ and "102382" into B$.

You can write files in a carefully controlled format using PRINT# USING. You also can use this option to control how many characters of a value are written to the file.

For example, suppose A$ = "LUDWIG", B$ = "VAN", and C$ = "BEETHOVEN". Then the statement:

```
PRINT# 1, USING"!.!.\␣␣\";A$;B$;C$
```

writes the data in nickname form:

```
L.V.BEET
```

(In this case, we did not want to add any explicit delimiters.) See PRINT USING for more information on the USING option.

PSET/PRESET/Graphics Statement

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

Draws a point on the display.

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

(*x*,*y*) specify the coordinates in which to draw the point. *X* is the horizontal coordinate and *y* is the vertical coordinate.

Color specifies the color of the point.

The only difference between the PSET and PRESET statements is the default values for *color*. If you use PSET, *color* defaults to the foreground color (1). If you use PRESET, *color* defaults to the background color (Ø).

> **Note:** BASIC does not print and does not issue an error message for points the coordinate values of which are beyond the edge of the screen. However, values outside the integer range (-32768 to 32767) cause an overflow error.

Sample Program

```
 5 CLS
1Ø FOR I=Ø TO 1ØØ
2Ø PSET (I,I)
3Ø NEXT I'draw a diagonal line to (1ØØ,1ØØ)
4Ø FOR I = 1ØØ TO Ø STEP -1
5Ø PRESET (I,I),Ø
6Ø NEXT I
7Ø 'clear the line by setting each pixel to Ø
```

Lines 1Ø to 3Ø draw a diagonal line on the screen from the home position to Position 1ØØ,1ØØ. Lines 4Ø to 6Ø erase the line by drawing another line at the same position in the background color.

PUT Statement

PUT [#]*buffer* [,*record number*]

Writes a record from a random buffer to a random access file.

Buffer is the number under which the file was opened. If *record number* is omitted, the record uses the next available record number (after the last PUT or GET). The largest possible record number is 16,777,215. The smallest record number is 1.

LSET, RSET, PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

Example

```
100 PUT 1, A$, B$, C$
```

QUIT

QUIT

Suspends BASIC and activates another application.

If this command is encountered at the command level, only the currently loaded program text is saved. In this case, it is impossible to CONT or RESUME this program. All open files are closed; all numeric variables are set to 0; and all string variables are set to null. When this work file is re-activated, the currently loaded program is still loaded.

If this command is encountered in a running program, the entire program is saved. When this work file is re-activated the program continues as if it had never been suspended.

RANDOMIZE Function

RANDOMIZE [*number*]

Reseeds the random number generator.

Number may be an integer, or single- or double precision number. If you omit *number*, BASIC suspends program execution and prompts you for a number before executing RANDOMIZE:

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

If the random number generator is not reseeded, the RND function returns the same sequence of numbers each time it is executed. To change the sequence of random numbers every time the RND function is executed, place a RANDOMIZE statement before the RND function.

Sample Program

```
10 CLS
20 RANDOMIZE
30 INPUT "PICK A NUMBER BETWEEN 1 AND 100"; A
40 B = INT(RND*100)
50 IF A=B THEN 80
60 PRINT "You lose, the answer is"B;"--try
again."
70 GOTO 20
80 PRINT "You picked the right number -- you
win."
```

READ Statement

READ *variable*[,*variable*,...]

Reads values from a DATA statement and assigns them to *variables*.

BASIC assigns values from the DATA statement on a one-to-one basis. The first time READ is executed, the first value in the first DATA statement is assigned to the first variable; the second time, the second value is assigned to the second variable, and so on.

A single READ may access 1 or more DATA statements, or several READs may access the same DATA statement. If a program contains multiple DATA statements, BASIC reads them in the order they appear.

The values read must agree with the variable types specified in a list of variables; otherwise, a "Syntax error" occurs.

If the number of variables in the READ statement exceeds the number of elements in the DATA statement(s), BASIC returns an "Out of DATA" error message. If the number of variables specified is less than the number of elements in the DATA statement(s), the next READ statements begin reading data at the first unread element.

To reread DATA statements from the start, use the RESTORE statement.

Example

```
READ T
```

reads a numeric value from a DATA statement and assigns it to variable T.

Sample Program

This program illustrates a common application for the READ and DATA statements.

```
40 PRINT "NAME", "AGE"
50 READ N$
60 IF N$="END" THEN PRINT "END OF LIST": END
70 READ AGE
80 IF AGE<18 THEN PRINT N$, AGE
90 GOTO 50
100 DATA "SMITH, JOHN", 30, "ANDERS, T.M.", 20
110 DATA "JONES, BILL", 15, "DOE, SALLY", 21
120 DATA "COLLINS, W.P.", 17, "END"
```

REM Statement

REM

Inserts a remark line in a program.

REM instructs the computer to ignore the rest of the program line, which lets you insert remarks in your program for documentation. Thus, when you look at a listing of your program, you can quickly interpret it.

If REM is used in a multistatement program line, it must be the last statement in the line.

You may use an apostrophe (') as an abbreviation for REM.

Sample Program

```
110 DIM V(20)
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
150 NEXT I
```

OR

```
110 DIM V(20)
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM + V(I)
140 NEXT I
```

RENUM Statement

RENUM [*new line*][,[*line*][,*increment*]]

Renumbers the program currently in memory. You can renumber the entire program or renumber from a specific line to the end.

Line is the line in the program where BASIC starts renumbering. If you omit *line*, it renumbers the entire program.

New line is the new line number assigned to *line*. If you omit *new line*, BASIC starts numbering at Line 10.

Increment tells BASIC how to number the successive line. If you omit *increment*, it increments by 10.

RENUM also changes all line number references appearing after GOTO, GOSUB, THEN, ON/GOTO, ON/GOSUB, ON ERROR GOTO, RESUME, and ERL.

You cannot use RENUM to change the order of program lines. For example, if a program has lines numbered 10, 20, and 30, the command RENUM 15,30 is illegal, since this would place Line 30 before Line 20.

Also RENUM cannot create line numbers greater than 65529. If you attempt to do this, BASIC returns an "Illegal function call" error and leaves the program unchanged.

If BASIC finds an undefined line number within the program, it prints a warning message, "Undefined line *xxxx* in *yyyy*," where *xxxx* is the undefined line number and *yyyy* is the line where it appears. RENUM renumbers the program despite this warning message. It does not change the incorrect line number reference, but it does renumber *yyyy*.

Examples

```
RENUM
```

renumbers the entire program, using an increment of 10. The new number of the first line is 10.

```
RENUM 600, 5000, 100
```

renumbers from Line 5000 to the end of the program. The first renumbered line becomes 600, and an increment of 100 is used between subsequent lines.

```
RENUM 100,,100
```

renumbers the entire program, starting with a new line number 100, and incrementing by 100s. Notice that the commas must be retained even though the middle argument is not used.

RESET Statement

RESET

Closes all open files on the disk drive.

If a disk contains any open files, RESET writes all blocks in memory to disk.

RESET ensures that all files on all diskettes are closed before you remove them from the drives. RESET is the same as a CLOSE statement for each open file.

RESTART ON/OFF/STOP Statement

RESTART ON
RESTART OFF
RESTART STOP

RESTART ON enables restart trapping
RESTART OFF disables restart trapping
RESTART STOP suspends restart trapping

These statements are used in conjunction with the ON RESTART GOSUB statement. (See the description of that statement for more information.)

RESTORE Statement

RESTORE [*line*]

Restores a program's access to previously read DATA statements.

Line is a line number that contains a DATA statement. If you specify *line*, the next READ statement accesses the first item in the specified DATA statement. If you omit *line*, BASIC resets to the first DATA statement in the program.

This lets your program reuse the same DATA lines.

Sample Program

```
160 READ X$
170 RESTORE
180 READ Y$
190 PRINT X$, Y$
200 DATA FIRST ITEM, SECOND ITEM
```

When you run this program, BASIC prints:

```
FIRST ITEM      FIRST ITEM
```

Because of the RESTORE statement in Line 170, the second READ statement starts over with the first DATA item.

RESUME Statement

RESUME [*line*]
RESUME NEXT

Resumes program execution after an error-handling routine.

RESUME without an argument and RESUME 0 both cause the computer to return to the statement in which the error has occurred.

RESUME *line* causes the computer to branch to the specified line number.

RESUME NEXT causes the computer to branch to the statement following the point at which the error has occurred.

Examples

```
RESUME
```

If an error has occurred, this line transfers program control to the statement in which it has occurred.

```
RESUME 10
```

If an error has occurred, transfers control to Line 10.

Sample Program

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230) AND(ERL=90) THEN PRINT "TRY
AGAIN" :RESUME 80
```

RETURN Statement

RETURN [*line*]

Returns control to the line immediately following the most recently executed GOSUB.

Line tells BASIC to return to a specific line in the program. If you omit *line*, BASIC goes to the line immediately following the GOSUB.

Use caution when specifying a line number with RETURN. Any other GOSUB, WHILE, or FOR statement remains active while a GOSUB subroutine is executing. If BASIC returns to a line number that is outside these loops, an error occurs because the loops were left incomplete.

If the program encounters a RETURN statement without execution of a matching GOSUB, an error occurs.

Example

```
RETURN 40
```

returns from the subroutine to Line 40 in the program.

Sample Program

```
330 PRINT "THIS PROGRAM FINDS THE AREA OF A
CIRCLE"
340 INPUT "TYPE IN A VALUE FOR THE RADIUS"; R
350 GOSUB 370
360 PRINT "AREA IS" ; A: END
370 A = 3.14 * R * R
380 RETURN
```

RIGHT$ Function

RIGHT$(*string,number*)

Returns the specified number of characters from the far right portion of *string*.

Number is an integer in the range 1 to 255.

If *number* is equal to or greater than the length of *string*, BASIC returns the entire string.

Examples

```
PRINT RIGHT$("WATERMELON", 5)
```

prints MELON.

```
PRINT RIGHT$("MILKY WAY", 25)
```

prints MILKY WAY.

Sample Program

```
850 RESTORE : ON ERROR GOTO 880
860 READ COMPANY$
870 PRINT RIGHT$(COMPANY$, 2), : GOTO 860
880 END
890 DATA "BECHMAN LUMBER COMPANY, SEATTLE, WA"
900 DATA "ED NORTON SEWER SERVICE, BROOKLYN, NY"
910 DATA "HAMMON MANUFACTURING COMPANY, HAMMOND,
IN"
```

This program prints the name of the state in which each company is located.

RND Function

RND[(*number*)]

Returns a random number in the range 0 and 1.

BASIC uses the current seed when generating a random number and produces the same sequence of random numbers each time the program is run unless you reseed the random number generator. Use the RANDOMIZE statement to reseed the random number generator.

If *number* is negative, RND starts the sequence of random numbers at the beginning. If *number* is 0, RND repeats the last number generated. If you omit *number* or specify a positive value, RND returns the next number in the sequence.

Example

```
PRINT RND(1)
```

prints the next decimal fraction in the sequence.

Sample Program

```
10 FOR I = 1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
```

This program produces 5 random integers. Line 20 converts the decimal fraction returned by RND to a real number and truncates the real number to an integer.

RSET Statement

RSET *field name = data*

Sets *data* in a direct access buffer *field name* in preparation for a PUT statement.

Field name is a string variable defined in a FIELD statement.

This statement is similar to LSET. The difference is that with RSET, data is right-justified in the buffer.

See LSET for details.

RUN Statement

RUN [*line*]
RUN *filename*[,R]

Executes a program.

Line is the program line where BASIC begins execution. If you omit *line*, BASIC executes the program from the beginning.

Filename specifies the file for BASIC to load into memory and execute.

If you specify the R option, BASIC does not close the open files before loading the new program into memory. If you omit the R option, BASIC closes all open files before loading the program.

RUN automatically clears all variables.

Examples

```
RUN
```

starts execution at the beginning of the program.

```
RUN 100
```

starts execution at Line 100.

```
RUN "program.a"
```

loads and executes *program.a*.

SAVE Statement

SAVE *filename* [,A]
SAVE *filename* [,P]

Saves a program in RAM or on disk with the specified name.

Filename is a standard file specification as described in Chapter 1. When you save a file, you must specify the filename. If the file already exists, its contents are lost when the file is re-created.

The A option tells BASIC to save the program in ASCII format. If you omit the A option, BASIC saves the file in a compressed format.

The compressed format takes less space than ASCII format. Also BASIC can save and load in compressed format faster than in ASCII format.

Use the ASCII format if you plan to use the MERGE command to merge the program with another. Also, data programs that will be read by other programs usually must be in ASCII.

When using the ASCII option, be sure your program has no embedded line feeds; otherwise, the computer will not be able to read it properly. Embedded line feeds are produced by typing [CTRL][J] in a program line.

For compressed-format programs, a useful convention is the extension *.bas*. For ASCII-format programs, use *.txt*.

> **Note:** If you do not specify an extension, BASIC automatically appends the default extension .BAS to the filename.

The P option protects the file by saving it in an encoded binary format. When a protected file is later run (or loaded), any attempt to list or edit it fails. The only operations that you can perform on a protected file are RUN, LOAD, MERGE, and CHAIN.

Examples

```
SAVE "A:file1.bas"
```

saves the resident program in compressed format as *file1.bas*. The file is placed on Drive A:.

```
SAVE "mathpak.txt", A
```

saves the resident program in ASCII form, using the name *mathpak.txt* in RAM.

SGN Function

SGN(*number*)

Determines *number*'s sign.

If *number* is a negative number, SGN returns -1.
If *number* is a positive number, SGN returns 1.
If *number* is zero, SGN returns 0.

Examples

```
Y = SGN(A * B)
```

determines the sign of the expression A * B, and passes the appropriate number (-1,0,1) to Y.

Sample Program

```
610 INPUT "ENTER A NUMBER"; X
620 ON SGN(X) + 2 GOTO 630, 640, 650
630 PRINT "NEGATIVE": END
640 PRINT "ZERO": END
650 PRINT "POSITIVE": END
```

SIN Function

SIN(*number*)

Returns the sine of *number*.

SIN returns the sine of the angle represented by *number*.

Number must be in radians. To obtain the sine of *number* when *number* is in degrees, use SIN(number * pi/180).

BASIC always returns the result as a double precision number.

Examples

```
PRINT SIN(7.96)
```

prints .99438531502814.

Sample Program

```
660 INPUT "ANGLE IN DEGREES"; A
670 PRINT "SINE IS"; SIN(A * .01745329)
```

SOUND Statement

SOUND *frequency,duration*

Generates a sound through the speaker.

Frequency is the desired frequency in hertz. This must be a positive integer in the range of 0 to 65535. The range of audible frequencies is approximately 94 to 15000.

Duration is the duration in clock ticks which occur 18.2 times per second. This must be a positive integer with a range of 0 to 2978.

If the duration is zero, any current SOUND statement that is running is turned off. If no SOUND statement is currently running, a SOUND statement with a duration of zero has no effect.

Example

```
30 SOUND RND*1000+37,2
```

This statement creates random sounds.

SPC — Function

SPC(*number*)

Prints *number* blanks.

Number is in the range 0 to 255.

You may use SPC only with PRINT, LPRINT, or PRINT# .

Example

```
PRINT "HELLO" SPC(15) "THERE"
```

prints:

```
HELLO               THERE
```

SQR Function

SQR(*number*)

Returns the square root of *number*.

Number must be greater than zero.

BASIC always returns the result as a double precision number.

Example

```
PRINT SQR(155.7)
```

prints 12.477980605852.

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 Statement

STOP

Stops program execution.

When BASIC encounters a STOP statement, it prints the message "BREAK IN *xxxx*," where *xxxx* is the line number that contains the STOP. STOP is primarily a debugging tool. During the break in execution, you can examine variables or change their values.

Use the CONT statement if you want to resume execution. If the program itself has been altered during the break, you cannot use CONT.

Unlike the END statement, STOP does not close files.

Sample Program

```
2260 X = RND(10)
2270 STOP
2280 GOTO 2260
```

A random number in the range 1 to 10 is assigned to X and then program execution halts at Line 2270. You can now examine the value X with PRINT X. Type CONT to start the cycle again.

STR$ Function

STR$(*number*)

Converts *number* to a string.

If *number* is positive, STR$ places a blank before the string. If *number* is negative, STR$ places a minus sign (-) before the string.

While arithmetic operations may be performed on *number*, only string functions and operations may be performed on the string.

The complementary function to STR$ is VAL.

Example

```
S$ = STR$(X)
```

converts the number X into a string and stores it in S$.

Sample Program

```
10 A = 1.6 : B# = A : C# = VAL(STR$(A))
20 PRINT "REGULAR CONVERSION" TAB(40) "SPECIAL
CONVERSION"
30 PRINT B# TAB(40) C#
```

SYSTEM Statement

SYSTEM

Returns you to the main menu.

BASIC closes all files before returning to the menu. Your resident BASIC program is lost, unless you first save it to RAM.

Examples

```
SYSTEM
```

Note: When you exit BASIC or a BASIC program with SYSTEM, no .BMI file of the program is created. When you exit a .BMI file with SYSTEM, the file is erased. If you exit BASIC with CTRL F10 or CTRL F9, you save the program you are working on in a .BMI file.

TAB Function

TAB(*number*)

Spaces to position *number* on the display.

Number must be in the range 1 to 255 and specifies the character position to which to tab. The leftmost position is 1, and the rightmost position is the set width minus 1.

If the current print position is already beyond space *number*, TAB goes to that position on the next line.

You cannot use TAB to move the cursor to the left.

You cannot use TAB more than once in a print list.

You may use TAB only with the PRINT and LPRINT statements.

Sample Program

```
10 PRINT "NAME" TAB(25) "AMOUNT":PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T.JONES","$25.00"
```

When you run this program, the display shows:

```
NAME                      AMOUNT

G.T.JONES                 $25.00
```

TAN Function

TAN(*number*)

Returns the tangent of *number*.

Return the tangent of the angle represented by *number*.

Number must be in radians. To obtain the tangent of *number* when it is in degrees, use TAN (number * pi/180).

BASIC always returns the result as a double precision number.

Example

```
PRINT TAN(7.96)
```

prints -9.3969620130791.

Sample Program

This programs asks you to input an angle in degrees and returns the tangent in radians.

```
720 INPUT "ANGLE IN DEGREES"; ANGLE
730 T = TAN(ANGLE * .01745329)
740 PRINT "TAN IS" T
```

TIME$ Statement

TIME$[= *string*]

Sets or retrieves the current time.

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

BASIC uses a 24-hour clock. For example, it sets 8:15 P.M. as 20:15:00.

Setting the Time

You set the time in the following format:

hh:*mm*:*ss*

The hours (*hh*) may be any number 0-23.
The minutes (*mm*) and the seconds (*ss*) may be any number 0 through 59.

If you omit the minutes, minutes **and** seconds default to zero. If you omit the seconds, seconds default to zero.

Although you may omit leading zeros in each of the values, you must include at least 1 digit of the preceding value. For example, you may type 1:5 to set the the time to 1:05 A.M. However, :5 is invalid.

Retrieving the Time

BASIC always returns the time in the 8-character (hh:mm:ss) format, with leading zeros. You may set the time before you enter BASIC.

Examples

```
TIME$ ="14:15"
```

sets the current time to 14:15:00.

```
TIME$ = "3:3:3"
```

sets the current time to 03:03:03.

```
A$=TIME$
```

assigns the current time to the variable A$.

```
PRINT TIME$
```

prints the current time.

TIMER/Trap Statement

TIMER *action*

Turns on, turns off, or temporarily halts timer event trapping.

Action may be any of the following:

ON	enables timer event trapping.
OFF	disables timer event trapping.
STOP	temporarily suspends timer event trapping.

Use the TIMER/Trap statement in a timer trap routine with the ON TIMER() GOSUB statement to detect when a specified period of time has elapsed.

The TIMER ON statement turns on the trap. BASIC checks the the value of timer after each program line. If the number is equal to that in the ON TIMER() GOSUB statement, BASIC transfers program control to the line number specified.

The TIMER STOP statement temporarily halts timer trapping. If the timer equals the specified number, BASIC does not transfer program control to the ON TIMER() GOSUB statement until you turn on trapping again by executing a TIMER ON statement. BASIC remembers that the timer value was equal and branches to the subroutine immediately after trapping is turned on again.

The TIMER OFF statement turns off timer trapping. BASIC does not remember if the value of timer equals the number specified when trapping is turned on again.

See ON TIMER() GOSUB for more information about timer event trapping.

Sample Program

See ON TIMER() GOSUB for an example.

TROFF, TRON — Statements

TROFF
TRON

Turn the trace function on/off.

TRON turns on the tracer and TROFF turns it off.

The tracer lets you follow program flow. This is helpful for debugging and for analyzing the execution of a program. After a program is debugged, you can remove the TRON and TROFF statements.

Each time the program advances to a new line, the tracer displays that line number inside a pair of brackets.

Sample Program

```
2290 TRON
2300 X = X * 3.14159
2310 TROFF
```

Lines 2290 and 2310 assure you that Line 2300 is actually being executed, because [2300] is printed on the display each time it is executed.

```
 5 TRON
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINT J;K;L
50 K=K+10
60 NEXT J
70 TROFF
80 END
```

When you run this program, BASIC prints:

```
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
```

VAL Function

VAL(*string*)

Calculates the numerical value of *string*.

VAL is the inverse of the STR$ function; it returns the number represented by the characters in a string argument. This number may be integer, single precision, or double precision, depending on the range of values and the rules used for typing all constants.

VAL terminates its evaluation on the first character that has no meaning in a numeric value.

If the string is nonnumeric or null, VAL returns a zero.

Examples

```
PRINT VAL("100 DOLLARS")
```

prints 100.

```
PRINT VAL("1234E5")
```

prints 123400000.

Sample Programs

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$) < 90000 OR VAL(ZIP$) > 96699
THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$) > 90801 AND VAL(ZIP$) <= 90815
THEN PRINT NAME$ TAB(25) "LONG BEACH"
```

This program searches for zip codes within the specified ranges to determine if they are within Long Beach or "out of state."

WRITE Statement

WRITE *data*[,*data*,...]

Writes data to the screen.

Data can be any string or numeric expression or variables. If you omit *data*, BASIC outputs a blank line.

The only difference between WRITE and PRINT is that WRITE prints commas between the data items and prints quotation marks around strings.

WRITE# Statement

WRITE#*buffer, data*[,*data*,...]

Writes *data* to a sequential access file.

Buffer is the number assigned to the file when you opened it.

Data may be numeric or string expressions. If you specify more than one data item, separate the items with commas.

WRITE# inserts commas between the data items it writes to the file. It delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters between the data.

WRITE# inserts a carriage return after writing the last data item to the file.

Example

```
A$="MICROCOMPUTER": B$="NEWS"
WRITE#1, A$,B$
```

writes the following image to a file:

```
"MICROCOMPUTER","NEWS"
```

TECHNICAL INFORMATION ABOUT LIBRARY FILES

Assembly Language Subroutines

Handheld BASIC programs can transfer control to assembly language subroutines using the CALL statement. These subroutines reside in specially formatted library files. The file DBCALLS.LIB is BASIC's standard library and contains supplied standard subroutines. You may add your own set of library routines by creating specially formatted library files and specify the name of this library with the LIBRARY statement. When searching for a subroutine, BASIC always searches libraries in the reverse order in which they were specified with the LIBRARY statement.

The LIBRARY Statement

LIBRARY *<library name>* is a string expression that indicates the name of an in-memory data file that is formatted as a library file which is used to search for subroutines specified in the CALL statement.

The LIBRARY CLOSE option lets you remove ALL libraries from the list of active libraries being searched by CALL statements. You may not remove only one library from the list.

The CALL Statement

CALL *<routine name>* [(*<argument list>*)] defines the name of the routine to which control is passed. All user libraries are searched for a routine with this name. If more than one LIBRARY statement is issued, the libraries are searched in the reverse order in which they were specified. If the subroutine is not found in any of the active libraries, an error occurs. *<argument list>* is an optional list of variables or constants, separated by commas, that are passed to the subroutine.

Invoking the CALL statement causes the following actions:

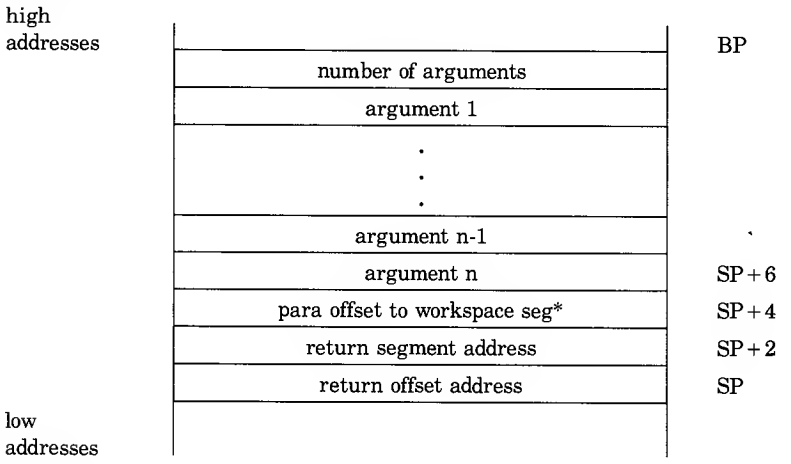
1. A search is made for the routine name in the active libraries (if any are defined).

2. Each argument in the argument list is evaluated and the proper value is pushed onto the stack. The arguments are evaluated and pushed in a left-to-right order, so the first argument in the argument list is in the stack at the highest memory address.

3. Control is passed to the subroutine by executing an 8086 far call to the proper segment/offset for the routine.

Calling Conventions

The state of the stack when the subroutine gains control is:



Note: *The workspace offset is the number of paragraphs below the current BASIC Data Segment (same as Stack Segment) where the library's workspace is located. This memory is preserved until either the library is closed, or BASIC's workspace file is deleted. *The initial size of this memory area is determined by an offset in the libraries header block.* This memory may be used to store global values unique to this invocation of the library. It is preserved across suspend operations so it may also be used to store information useful in restarting the library — i.e. file control blocks.

To access this workspace, one must subtract the specified number of paragraphs from BASIC's DS.

The following rules must be adhered to in assembly level subroutines:

1. When the routine begins execution, the values of DS, ES, and SS are set to the segment address of BASIC's data segment. When the routine exits, all three segments MUST still point to BASIC's data segment. However, because any file I/O operations may move BASIC's data segment, special care must be taken to preserve this address. All values in segment registers are correctly updated by the operating system if the file system moves. BASIC's DATA segment address MUST remain in the SS register during any O/S calls since a file system request may force BASIC to return some free space to the file system in order to complete the I/O request.

   Note that if this occurs, BASIC's stack will move so that it is quite possible that when you return from an O/S call, the SP will have changed. All of the data on the stack (i.e. return addresses and arguments) will still be in the same relative locations but any stack markers you have stored must be adjusted.

2. If an OS call forces BASIC to return some Free space to the file system, the stack pointer offset is changed during the OS call. All values on the stack are adjusted correctly.

3. If interrupts are disabled in the routine, they must be enabled before returning to BASIC.

4. The stack must be in a consistent state when the routine exits. That is, the stack pointer (SP), when BASIC regains control must point to the workspace segment offset parameter (input SP+4). The most convenient way to do this is to return from the routine using a 8086 far return instruction.

5. The stack passed to the routine may be used for temporary variables. If global (to the library) variables need to be allocated, they may be either allocated in the library's code segment (assuming the library is in RAM) or the library may define and use a reserved block of memory.

6. The values of any argument passed by reference may be changed. However, care must be taken to properly format the data using the information in the INTERNAL DATA FORMATS section as a guide.

Warning: If the argument is a string literal in the program, the string descriptor points to program text. If this string is modified, the actual program is modified. To avoid string literals in the program, concatinate a null string (" ") to the literal in the program. For example, use

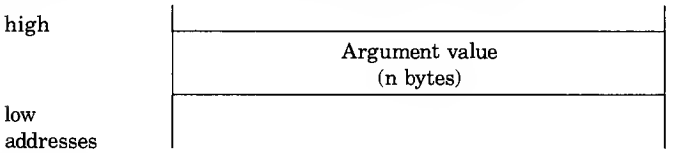
```
20 A$ = "ABCD"+""
```

This forces the string literal to be copied into string space. The string may then be modified without affecting the program.

7. The contents of a string may be changed, but care must be used in modification of the string descriptor. The string length may be decreased, but not increased.

Argument Values on the Stack

The format of the values pushed on the stack for an argument varies depending upon the argument type. All argument values, however, have the same generalized format:



All pointers on the stack are relative to BASIC's data segment which is contained in DS when the routine is called.

The Argument length byte defines the byte length of the argument value. This is useful in that the address of the argument value minus this length results in the address of the type/length word for the next argument on the stack.

The value of the argument type byte is a bit-encoded value with the following bit definitions:

Bit 7 Reference/value bit. If this bit is 0, the argument was passed by reference. In this case, the argument value will be the address of the value or a descriptor (for strings and arrays). If this bit is 1, the argument was passed by value. In this case, the argument value is the actual value of the argument.

Bit 5 — Null argument bit. If this bit is set, a null argument is passed. This is used as a place keeper for calls with optional arguments but strict argument ordering (i.e. "CALL FOO(A,B, ,C)"). There is no argument value for this type of argument (length is Ø).

Bit 4 — Array reference bit. If this bit is 1, the argument is an array passed by reference (i.e. – "ARRAY()"). The argument value is a pointer to an array descriptor. Bits Ø-3 of this byte define the data type of the array elements.

Bit 3 — Double precision floating bit. If this bit is set, the argument is a double precision floating point number.

Bit 2 — Single precision floating bit. If this bit is set, the argument is a single precision floating point number.

Bit 1 — Integer bit. If this bit is set, the argument is a 16-bit integer.

Bit Ø — String bit. If this bit is set, the argument value is a pointer to a string descriptor.

For bits 1-3, if the reference/value bit (bit 7) is Ø, the argument value is a pointer to the value. If bit 7 is set, the argument value is the actual value of the argument.

All simple variable parameters are passed by reference. Arrays, however, are passed in two different ways. If a subscript is given with the array, it is evaluated and the address of the specific array element is passed by reference. If a null subscript (i.e. – "FOO()") is given, the address of the array descriptor is passed. The array must already exist for this form to be used. All expressions and constants (except for strings) are passed by value.

Returning Errors to BASIC

When BASIC regains control after a call, the value of the 8086 flag-word is significant. The carry bit in the flag word is used to indicate whether or not the called subroutine encountered an error. If the carry bit is clear (a zero) upon return, BASIC assumes there were no errors and continues executing statements normally. However, if the carry bit is set, BASIC returns an error condition. In this case, the contents of the AL register is assumed to contain a BASIC error code. Control is then passed to BASIC's error processing code to process the error passed in AL. This will have the same effect as the user executing an ERROR statement with this error code. If error-trapping is enabled, the GOSUB specified is taken. The error code may be determined by using the ERR variable.

Internal Data Formats

The internal storage format of a data item within BASIC is:

Integer

The value is stored as a 16-bit 2's complement integer. 8086 convention stores the least significant byte first, and then the most significant byte.

Single Precision Floating

A single precision number is a 4-byte value in decimal floating point format. This format is:

Byte: Exponent. The exponent is broken into the following parts:

Bit 7 : Sign bit. This bit is 0 if the number is positive and 1 if it is negative.

Bits 6-0 : Exponent value in excess-64 format. This value is a power of 10 exponent which determines the decimal position in the represented number.

3 Bytes : Mantissa. These bytes contain pack BCD digits with the most significant digit of the number in the byte of the mantissa with the lowest address. This mantissa can contain 6 digits (2 per byte).

NOTE: Excess-64 format can be converted to decimal by subtracting 64 from bits 6-0 of the exponent. Conversely, to convert to excess-64 format, add 64 to the number. For example:

Exponent	Excess 64
5	69 (45 Hex)
−16	48 (30 Hex)

An excess-64 format exponent of 0 is reserved to indicate that the value represented is zero. Thus, valid exponents are in the range −63 to +63.

Double Precision Floating

A double precision number is stored as an 8-byte value. The format if this value is the same as for single precision except that there are 7 bytes of mantissa rather than 3. The most significant byte (lowest address) is still the exponent byte.

String

Strings are defined by a 3-byte string descriptor. The format of this descriptor is:

Byte : 1 byte unsigned integer that contains the length of the string.

Word : Two byte pointer that contains the address of the first byte of the string.

Array Descriptor

An array descriptor is a variable size block which contains information about the array. Its format is:

Byte : Number of dimensions in the array
Word : Number of entries for dimension #1
Word : Number of entries for dimension #2
.
.
Word : Number of entires for last dimension

The byte following the array descriptor is the first byte of the first array element.

Library File Format

In order for BASIC to use a library file, its format must match the following description.

Header Block

The library file header block starts with the first word of the file. The format of a header block is:

Word : Revision number. This number is used by BASIC to determine if this library file is compatible with the current version of BASIC. Library files adhering to the definition defined here must contain a Ø in the high byte of this word and a 1 in the low byte.

Word : Memory requirements. If this word contains a non-zero value, it is interpreted as the number of bytes of reserved memory allocated for this library. This memory is allocated using the OS "get workspace" facility. The segment identifier is passed to all routines on the stack and the value passed to each routine on the stack will be −1.

Word : Initialization routine address. This address is the offset within the library file where control is passed when the library file is opened. This routine may perform any initialization that is needed. When the initialization routine gains control, DS:DX will point to the following structure:

Byte : BASIC type. This is one of the following:
1 – for Handheld BASIC
2 – for GW-BASIC
3 – for BASIC in C (no windows)
4 – for Windows-BASIC

Byte : BASIC version. For example, 1 in version 1.Ø2.

Byte : BASIC revision. For example, 2 in version 1.Ø2.

Byte : Math pack that is being used. This will be Ø for the decimal math pack or 1 for the binary math pack. (Handheld BASIC uses the decimal math pack.)

This structure lets the library initialization routine determine if it is compatible with the BASIC that is attempting to use it. If the initialization code determines that it can not be used by this BASIC, it sets the carry flag, sets the value of AL to an appropriate error code and returns to BASIC. The contents of this structure must not be modified.

Word : Termination routine address. This address is an offset within the library file where control is passed when the library file is closed. This routine may perform any termination and cleanup operations needed.

Word : Suspend routine address. This address is the offset to the routine which is called whenever BASIC suspends itself due to a QUIT command or the quit/run-previous keys being typed. It is the responsibility of this routine to save the current state of the library so that it may be restored when BASIC is restarted. Some examples of things that should be saved are:

1. Since all files are closed on a suspend, information needed to re-open files must be saved.

2. If the library is manipulating any devices, the state of the device must be saved.

Word : Restart routine address. This routine is called when BASIC is restarted after a suspend.

Note: If the address of any of the above routines is zero, no call is made.

Routine Dispatch Table

This table must immediately follow the header block. It contains a list of pointers to routine name blocks. It is REQUIRED that the routine names are sorted in descending ASCII sequence. Thus the first pointer in the table must point to the lowest sorted routine name and the last entry point to the highest sorted name. The format of this table is:

Word : The number of routines in the dispatch table (n).

n Words : Pointers to the routine name blocks.

Routine Name Blocks

These variable size blocks are used to link the name of the routine with the offset of the start of the routine. Each routine in the library must have a name dispatch block in order for BASIC to find it. The format of a name dispatch block is:

Word : Routine dispatch address. This is the offset within this library file of the start of the routine.

Byte : Number of bytes in the routine name (n)

n Bytes : Routine name. The name must be stored as a string of bytes in UPPER CASE characters.

Routine Code

The rest of the library file contains the code for all the routines in the library. Word 0 of each routine name block must point to the first instruction of the appropriate routine.

BASIC ERROR CODES AND MESSAGES

Number	Code	Message
1	NF	**NEXT without FOR** BASIC executed a NEXT statement without previously executing a FOR statement, or a variable in a NEXT statement does not correspond to a previously executed FOR statement.
2	SN	**Syntax error** BASIC encountered a line that contains an incorrect sequence of characters (such as unmatched parentheses, misspelled statement, incorrect punctuation, etc.).
3	RG	**RETURN without GOSUB** BASIC executed a RETURN statement without previously executing a GOSUB statement.
4	OD	**Out of DATA** When executing a READ statement, BASIC could not find any DATA statements or unread data items.

Number	Code	Message
5	FC	**Illegal function call**

A parameter that is out of range was passed to a math or string function. This error may also occur as the result of:

- negative array subscript or an unreasonably large array subscript.
- negative or zero argument with LOG.
- negative argument to SQR.
- negative mantissa with a noninteger exponent.
- invalid exponential number.
- improper argument to MID$, LEFT$, RIGHT$, TAB, SPC, SPACE$, STR$, or LEFT$, RIGHT$, TAB, SPC, STR$ or ON...GOTO.
- negative record number used with GET or PUT.

6 OV Overflow

The result of a calculation was too large to be represented in BASIC numeric format. If underflow occurs, the result is zero, and execution continues without an error.

7 OM Out of memory

A program is too large, has too many FOR loops or GOSUBs, has too many variables, or has expressions that are too complicated.

8 UL Undefined line number

A nonexistent line was referenced in a GOTO, GOSUB, or IF...THEN...ELSE statement.

9 BS Subscript out of range

An array element is referenced with a subscript outside the dimensions of the array or with the wrong number of subscripts.

Number	Code	Message
10	DD	**Redimensioned Array/Duplicate Definition** BASIC encountered 2 DIM statements for the same array, or a DIM statement after the default dimension of 10 had already been established for that array.
11	/0	**Division by zero** An expression includes division by zero, or the operation of involution results in zero being raised to a negative power. BASIC supplies machine infinity with the sign of the numerator as the result of the division, or it supplies positive machine infinity as the result of the involution.
12	ID	**Illegal direct** A statement that is illegal as a command was entered at BASIC's prompt.
13	TM	**Type mismatch** A string variable name was assigned a numeric value or vice versa. A string function was given a numeric argument or vice versa.
14	OS	**Out of string space** The amount of memory used by string variables exceeded the amount of free memory.
15	LS	**String too long** An attempt was made to create a string more than 255 characters.
16	ST	**String formula too complex** A string expression is too long or too complex. The expression should be broken into smaller expressions.

Number	Code	Message
17	CN	**Can't continue** An attempt was made to continue a program that: • halted because of an error. • was modified during a break in execution. • does not exist.
18	UE	**Undefined error.**
19	NR	**No RESUME** BASIC executed an error-handling routine that did not have a RESUME statement.
20	RW	**RESUME without error** BASIC executed a RESUME statement when no error had occurred.
21	UE	**Undefined error**
22	MO	**Missing operand** BASIC encountered an expression that contained an operator but no operand.
23	BO	**Line buffer overflow** The line being input is too long.
24	DT	**Device Timeout** BASIC did not receive information from an I/O device within a predetermined amount of time.
25	DF	**Device Fault** An incorrect device designation has been entered.
26	FN	**FOR without NEXT** BASIC executed a FOR statement that did not have a matching NEXT.

Number	Code	Message
27	OP	Printer Error
28	UE	Undefined Error
29	UE	Undefined Error
30	UE	Undefined Error

Disk Errors

Number	Code	Message
50	FO	**FIELD overflow**
		A FIELD statement is allocating more bytes than the specified record length of the direct access file.
51	IE	**Internal error**
		An internal malfunction has occurred in BASIC. Report to Radio Shack the conditions under which the message appeared.
52	BN	**Bad file number**
		BASIC encountered a reference to a buffer number that is not open or is out of the range of the number of files specified when BASIC was loaded.
53	FF	**File not found**
		A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.
54	BM	**Bad file mode**
		An attempt was made to use PUT, GET, or LOF with a sequential file, to LOAD a direct file.
55	AO	**File already open**
		BASIC encountered an OPEN statement for sequential output, or a KILL statement, for a file that is already open.

Number	Code	Message
57	IO	**Device I/O Error**
		An Input/Output error occurred. This is a fatal error; the operating system cannot recover it.
58	FE	**File already exists**
		The filename specified in a NAME statement is identical to a filespec already in use on the disk.
61	DF	**Disk full**
		All disk storage space is in use.
62	EF	**Input past end**
		BASIC executed an INPUT statement after all the data in the file had been read, or BASIC executed an INPUT statement to a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.
63	RN	**Bad record number**
		In a GET or PUT statement, the record number is either greater than the maximum allowed (16,777,215) or equal to zero.
64	NM	**Bad file name**
		An illegal filename was used with a LOAD, SAVE, KILL, or OPEN statement (for example, a filename with too many characters).
66	DS	**Direct statement in file**
		Information in a non-ASCII format was encountered while LOADing an ASCII-format file. The LOAD is terminated.
67	FL	**Too many files**
		The diskette already contains the maximum number of files allowed. This usually occurs on SAVE or OPEN. An attempt was made to create a new file (using SAVE or OPEN) when all directory entries are full.

Number	Code	Message
68	**DU**	**Device Unavailable**
		An attempt was made to open a file to a nonexistent device. It may be that hardware does not exist to support the device, such as LPT2: or LPT3:, or that the device is disabled.
69	**UE**	**Undefined Error**
73	**AF**	**Advanced Feature**
75	**FA**	**File Access Error**
77	**UE**	**Undefined Error**
89	**BL**	**Bad Library Format**
90	**UR**	**Undefined Library Routine**
91	**LF**	**Illegal DBMS Call**
92	**IC**	**Illegal Argument Count**
93	**IT**	**Illegal Argument Type**
94	**IV**	**Illegal Argument Value**

DBCALLS.LIB ERRORS (via DBERRORS (x))

(O/S Specific Errors — Errors not related specifically to the O/S Data Base support)

Number	Message
2	File Not Found
4	Too Many Open Files
5	File Access Denied
6	Invalid File Handle
8	Out of Memory
9	Invalid Memory Block
11	Bad File Format
12	Invalid Access Request
18	No More Files
20	File Too Big
21	Internal File System Error
22	Bad File Name
24	General I/O Error
26	File Checksum Error
28	Invalid Time
29	Invalid Date

(Error Codes From Database O/S Calls. O/S errors specific to the Database O/S calls)

Number	Message
64	**Record Already Opened**
65	**No Opened Record**
66	**Record Not Found**
67	**Field Not Found**
68	**Too Many Fields**
69	**No Fields Defined**
70	**Uninitialized Data**
71	**Bad Field Data Size**
72	**Field Already Exists**
73	**Bad Sort Key Specified**
74	**Query Buffer Overflow Error**
75	**Bad Field Type Specified**
76	**Too Many Records**
77	**Record Too Big**

ASCII CHARACTER CODES

The table in this appendix lists the characters generated by ASCII codes. (Note: All ASCII codes in this table are expressed in **decimal** form.)

You can display the characters listed by using the BASIC statement PRINT CHR$(*code*), where *code* is the ASCII code.

For Codes 0-31, the table also lists the standard interpretations. The interpretations are usually used for control functions.

> **Note:** The BASIC program editor has its own special interpretation of some codes and may not display the character listed.

ASCII CHARACTER CODES

ASCII Code	Character	Control Character
000	(null)	NUL
001	☺	SOH
002	☻	STX
003	♥	ETX
004	♦	EOT
005	♣	ENQ
006	♠	ACK
007	•	
008	◘	BS
009	(tab)	HT
010	(line feed)	LF
011	♂	
012	(home)	VT
013	(carriage return)	CR
014	♫	SO
015	☼	SI
016	►	DLE
017	◄	DC1
018	↕	DC2
019	!!	DC3
020	¶	DC4
021	§	NAK
022	▬	SYN
023	↨	ETB
024	↑	CAN
025	↓	EM
026	→	SUB
027	←	ESC
028	∟	FS
029	↔	GS
030	▲	RS
031	▼	US

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
032	(space)	068	D
033	!	069	E
034	"	070	F
035	#	071	G
036	$	072	H
037	%	073	I
038	&	074	J
039	'	075	K
040	(	076	L
041	)	077	M
042	*	078	N
043	+	079	O
044	,	080	P
045	-	081	Q
046	.	082	R
047	/	083	S
048	0	084	T
049	1	085	U
050	2	086	V
051	3	087	W
052	4	088	X
053	5	089	Y
054	6	090	Z
055	7	091	[
056	8	092	\
057	9	093	]
058	:	094	^
059	;	095	_
060	<	096	`
061	=	097	a
062	>	098	b
063	?	099	c
064	@	100	d
065	A	101	e
066	B	102	f
067	C	103	g

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
104	h	140	î
105	i	141	ì
106	j	142	Ä
107	k	143	Å
108	l	144	É
109	m	145	æ
110	n	146	Æ
111	o	147	ô
112	p	148	Ö
113	q	149	ò
114	r	150	û
115	s	151	ù
116	t	152	ÿ
117	u	153	Ö
118	v	154	Ü
119	w	155	¢
120	x	156	£
121	y	157	¥
122	z	158	Pt
123	{	159	ƒ
124	¦	160	á
125	}	161	í
126	~	162	ó
127	SPACE	163	ú
128	Ç	164	ñ
129	ü	165	Ñ
130	é	166	ª
131	â	167	º
132	ä	168	¿
133	à	169	⌐
134	å	170	¬
135	ç	171	½
136	ê	172	¼
137	ë	173	¡
138	è	174	«
139	ï	175	»

ASCII CHARACTER CODES

ASCII Code	Character	ASCII Code	Character
176	▒	212	╘
177	▒	213	╒
178	▒	214	╓
179	│	215	╫
180	┤	216	╪
181	╡	217	┘
182	╢	218	┌
183	╖	219	▉
184	╕	220	▄
185	╣	221	▌
186	║	222	▐
187	╗	223	▀
188	╝	224	α
189	╜	225	β
190	╛	226	Γ
191	┐	227	π
192	└	228	Σ
193	┴	229	σ
194	┬	230	μ
195	├	231	τ
196	─	232	Φ
197	┼	233	Θ
198	╞	234	Ω
199	╟	235	δ
200	╚	236	∞
201	╔	237	Ø
202	╩	238	∈
203	╦	239	∩
204	╠	240	≡
205	═	241	±
206	╬	242	≥
207	╧	243	≤
208	╨	244	⌠
209	╤	245	⌡
210	╥	246	÷
211	╙	247	≈

ASCII CHARACTER CODES

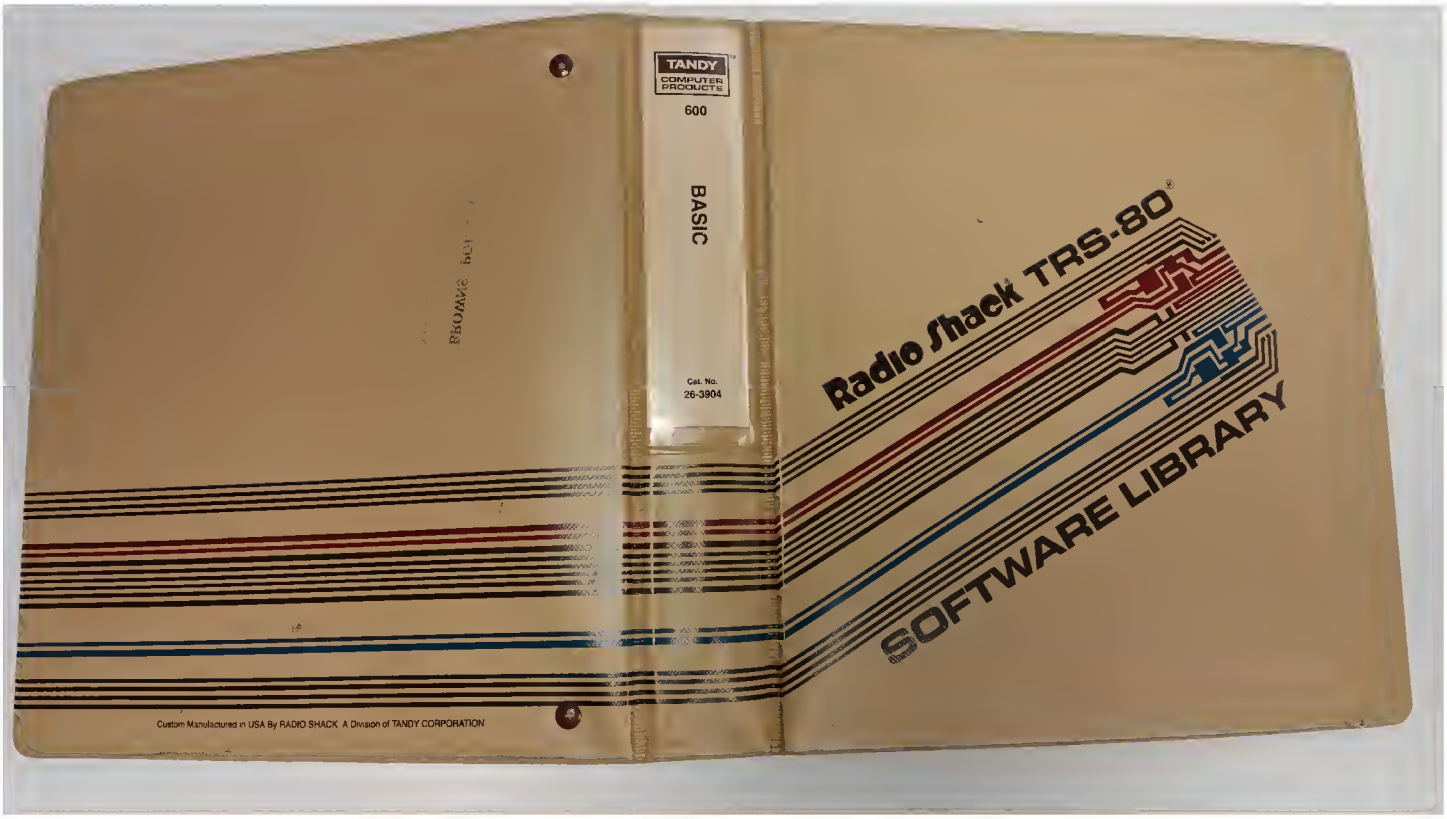
ASCII Code	Character	ASCII Code	Character
248	°	252	η
249	+	253	²
250	+	254	■
251	√	255	(blank 'FF')

INDEX

RADIO SHACK
A Division of Tandy Corporation
Fort Worth, Texas 76102


Radio Shack® TRS-80®
SOFTWARE LIBRARY


Custom Manufactured in USA By RADIO SHACK, A Division of TANDY CORPORATION


TANDY
COMPUTER PRODUCTS
600
BASIC
Cat. No.
26-3904
Radio Shack TRS-80®
SOFTWARE LIBRARY
Custom Manufactured in USA By RADIO SHACK A Division of TANDY CORPORATION