0319P TI-99/2 Book 4 BASIC Reference Guide (FINAL DRAFT)

Cover Copy: A comprehensive guide to every command, statement, and function in TI-99/2 BASIC.

Basic Computer 99/2
Book 4: BASIC Reference Guide

Book 4: BASIC Reference Guide

INTRODUCTION

This manual provides a complete explanation of all the commands, statements, and functions in the TI-99/2 BASIC language built into your computer. After you've gained proficiency in programming, this guide serves as your primary reference for TI-99/2 BASIC commands, statements, and functions.

Notational Conventions

The discussion for each command, statement, or function begins with a line that shows its general format, following these notational conventions.

;§ §;	Braces indicate a choice of items. You may use only one of the items enclosed in braces.
[ ]	Brackets indicate optional items . You may use the items if you wish, but they are not required.
. . .	An ellipsis indicates that the preceding item may be repeated as many times as you desire.
italics	Italicized words indicate the kind of item or items to be used. Enter your own choice in place of the italicized words when you enter the statement or command.

Examples

For each statement or command in this manual, program examples are shown on the following page. Each line that you must enter is indicated by the prompt character (>) to the left of the line, just as it appears on the screen. Lines which the computer places on the screen do not show the prompt character.

COMMANDS

When the prompt and cursor appear in the lower-left-hand corner of the screen, your Basic Computer 99/2 is in Command (Immediate) Mode, and you may enter a command. Commands are not preceded by line numbers; the computer performs the task immediately.

Some commands may also be used as statements in programs, as noted in the discussions.

NEW

NEW

The NEW command erases the program that is currently stored in memory and cancels any BREAK or TRACE command in effect. NEW also closes any open files and erases all variable values and the table in which variable names are stored.

After the NEW command is performed, the screen is cleared and the message TI-99/2 BASIC READY is displayed on the screen. The prompt and flashing cursor (>_) indicate that you may enter another command or a program line.

BYE

BYE

The BYE command closes all open files, erases the program and all variables in memory, and resets the computer, causing the master title screen to reappear. To leave TI-99/2 BASIC, always use the BYE command instead of the QUII key combination because QUII does not close open files.

```
NEW
```

(The screen is cleared, and the following appears at the bottom of the screen.)

```
TI-99/2 BASIC READY
>_
```

```
>NEW

>100 X$="HELLO, GENIUS!"
>110 PRINT X$
>RUN
 HELLO, GENIUS!

 ** DONE **

>BYE
```

(The master title screen appears.)

LIST

LIST [line-list]
LIST "HEXBUS.device-number"[:line-list]

The LIST command is used to print or display the program lines in memory. If you enter the LIST command without a line-list, then the entire program is printed or displayed. The program lines are always listed in ascending order, without unnecessary blank spaces.

If you enter HEXBUS.device-number, the program lines are printed on the specified device. Device-numbers for HEX-BUS™ peripherals are listed on page XX.

If you enter a line-list, it may consist of a single number, a single number preceded by a hyphen, a single number followed by a hyphen, or a hyphenated range of line numbers.

Command	Lines Displayed or Printed
LIST	All program lines
LIST x	Program line number x
LIST x-y	Program lines between x and y, inclusive
LIST x-	Program lines greater than or equal to x
LIST -y	Program lines less than or equal to y

You can stop any listing by pressing BREAK or CLEAR.

If there is a program in memory and the line-list specifies a line number that is not in the program, the following conventions apply.

- For line numbers greater than any in the program--the highest-numbered program line is listed.
- For line numbers less than any in the program--the lowest-numbered program line is listed.
- For line numbers between lines in the program--the next higher-numbered line is displayed.

You can use LIST to direct output to a peripheral device. For example,

```
LIST "HEXBUS.10"
```

causes your program to be printed, if the Printer/Plotter is attached, and

```
LIST "HEXBUS.20":100-200
```

outputs program lines 100 through 200 to the RS232 Interface. Note that HEXBUS and the number of the device must be enclosed in quotation marks. For more information about device-numbers used with the LIST command, refer to the owner's manual that comes with the peripheral device.

```
>NEW

>100 A=279.3
>120 PRINT A;B
>110 B=-456.8
>LIST
 100 A=279.3
 110 B=-456.8
 120 PRINT A;B


>LIST 110
 110 B=-456.8


>LIST 90-120
 100 A=279.3
 110 B=-456.8
 120 PRINT A;B


>LIST 110-
 110 B=-456.8
 120 PRINT A;B


>LIST -110
 100 A=279.3
 110 B=-456.8


>LIST 150-
 120 PRINT A;B

>LIST -90
 100 A=279.3

>LIST 105
 110 B=-456.8
```

If you enter a LIST command and specify a line number that is less than 1 or greater than 32767, the message BAD LINE NUMBER is displayed.

If you specify a line number which is not an integer, the message INCORRECT STATEMENT is displayed.

If no program is in memory when you enter a LIST command, the message CAN'T DO THAT is displayed.

RUN

RUN [line-number]

The RUN command causes the computer to begin executing the program stored in memory. Before the program starts running, the computer

- sets the values of all numeric variables to zero
- sets the values of all string variables to a null string (one containing no characters)
- checks for certain program errors (see Appendix XX)
- closes any open files.

If no line-number is specified when the RUN command is entered, the computer starts program execution at the lowest-numbered line in the program.

If you specify a line-number, the program starts running at the specified program line. Note in this example that because the program begins running at line 110, the value of A remains zero.

If you specify a line-number that is not in the program, the message BAD LINE NUMBER is displayed.

If you enter a RUN command when there is no program in memory, the message CAN'T DO THAT is displayed.

```
>NEW

>100 A=-16
>110 B=25
>120 PRINT A;B
>RUN
 -16  25

 ** DONE **

>RUN 110
  0  25

 ** DONE **
```

EDIT

EDIT line-number
line-number FCTN E
line-number SHIFT E
line-number FCTN X
line-number SHIFT X

You can change existing program lines in Edit Mode. To enter Edit Mode, type the EDIT command followed by a line-number, or type a line-number followed by ECIN E (UP ARROW) or ECIN X (DOWN ARROW). SHIFT can be substituted for ECIN in these operations. If you specify a line-number that is not in the program, the message BAD LINE NUMBER is displayed.

Entering Edit Mode displays the line specified by the line-number. The prompt character (>) is not displayed. The cursor is positioned in the second character position to the right of the line number. You can change any character on the line (except the line number) by using the special keys described below and typing over the characters you wish to change.

ECIN_S or SHIFT_S (LEFT ARROW)--The LEFT ARROW (backspace) key moves the cursor one position to the left. When the cursor moves over a character, it does not delete or change the character.

ECIN_D or SHIFT_D (RIGHT ARROW)--The RIGHT ARROW (forwardspace) key moves the cursor one position to the right. When the cursor moves over a character, it does not change or delete the character.

ECIN_2 (INS)--The INSert key works the same in Edit Mode as it does in Command Mode. See "Tour of the Keyboard" in Book 1.

ECIN_1 (DEL)--The DELete key works the same in Edit Mode as it does in Command Mode. See "Tour of the Keyboard" in Book 1.

ECIN_4 (CLEAR)--The CLEAR key scrolls the current line up on the screen and leaves the program line unchanged. The computer then leaves Edit Mode.

ECIN_3 (ERASE)--The ERASE key erases the entire text of the program line currently displayed. The line number is not erased.

ENTER--The ENTER key replaces the program line in memory with the edited (displayed) line, and the computer leaves Edit Mode. Note that the cursor does not have to be at the end of the line for the entire line to be entered. If you erase the entire text of the program line and then press ENTER, the program line is deleted.

ECIN_E or SHIFT_E (UP ARROW)--The UP ARROW key replaces the program line in memory with the edited (displayed) line and then displays the next lower-numbered line in the program. If no lower-numbered program line exists, the computer leaves the Edit Mode. Note that the cursor does not have to be at the end of the line for the entire line to be entered by the UP ARROW key.

ECIN_X or SHIEI_X (DOWN ARROW)--The DOWN ARROW key replaces the program line in memory with the edited (displayed) line and then displays the next higher-numbered line in the program. If no higher-numbered program line exists, the computer leaves the Edit Mode. Note that the cursor does not have to be at the end of the line for the entire line to be entered by the DOWN ARROW key.

NUMBER

NUMBER
[initial-line][,increment]
NUM

When you enter the NUMBER command, the computer enters the Number Mode and automatically generates line numbers for your program. If no initial-line and no increment are specified, the initial-line is 100 and the increment is 10.

If you include an initial-line and an increment, the first line number displayed is the specified initial-line. Succeeding line numbers are generated by adding the specified increment to the current line number.

If you specify only an initial-line, 10 is used as the increment.

If you specify only an increment, 100 is used as the initial-line. Note the comma before the 5 in the example; to specify only an increment, precede the increment with a comma.

To stop the automatic generation of line numbers and leave Number Mode, press ENTER immediately after the generated line number is displayed. The empty line is not added to the program.

If a line number generated by the NUMBER command is already a line in the program, the existing program line is displayed with the line number. The prompt character (>) is not shown to the left of the line number, indicating that the line is an existing program line and you may edit the line. If you do not want to change the existing line, press ENTER when the line is displayed. The line is entered as is, and the next line number is generated.

If you enter a program line with an error, the appropriate error message is displayed, and the same line number is displayed again. Retype the line correctly and then enter it again.

If the next line number to be generated in Number Mode is greater than 32767, the computer leaves Number Mode.

```
>NEW

>NUM
>100 B$="HELLO!"
>110 PRINT B$
>120 (Press ENTER to exit Number Mode.)

>NEW

>NUMBER 10,5
>10 C=38.2
>15 D=16.7
>20 PRINT C;D
>25 (Press ENTER to exit Number Mode.)
>LIST
>10 C=38.2
>15 D=16.7
>20 PRINT C;D

>NEW

>NUMBER 50
>50 C$="HI!"
>60 PRINT C$
>70 (Press ENTER to exit Number Mode.)

>NEW

>NUM ,5
>100 Z=99.7
>105 PRINT Z
>110 (Press ENTER to exit Number Mode.)

>NEW
>100 A=37.1
>110 B=49.6
>NUMBER 110
>110 B=49.6
>120 PRINT A;B
>130 (Press ENTER to exit Number Mode.)
 LIST
 100 A=37.1
 110 B=49.6
 120 PRINT A;B
```

Editing in Number Mode

In Number Mode, all of the editing keys may be used whether you are entering new lines or changing existing program lines. Some of the keys work differently in Number Mode than in Command Mode.

ENTER

- If you press ENTER immediately after a new line number is generated, the computer leaves Number Mode.

- If you type a statement after the line number is generated and then press ENTER, the new line is added to the program. The next line number is then generated.

- If you do not edit a program line before pressing ENTER, the line remains the same in the program. The next line number is then generated.

- If you erase the entire text of an existing program line (leaving only the line number on the screen) and then press ENTER, the computer leaves Number Mode. The program line is not removed from the program.

- If you edit an existing program line and then press ENTER, the existing program line is replaced by the edited line. The next line number is then generated.

CTRL E or SHIFT E (UP ARROW)--The UP ARROW key works exactly the same as the ENTER key in Number Mode.

CTRL X or SHIFT X (DOWN ARROW)--The DOWN ARROW key works exactly the same as the ENTER key in Number Mode.

CTRL S or SHIFT S (LEFT ARROW)--The LEFT ARROW key moves the cursor to the left. When the cursor moves over a character it does not delete or change the character.

CTRL D or SHIFT D (RIGHT ARROW)--The RIGHT ARROW key moves the cursor to the right. When the cursor moves over a character, it does not delete or change the character.

CTRL 2 (INS)--The INSert key works the same in Number Mode as it does in Command Mode. See "Tour of the Keyboard" in Book 1.

CTRL 1 (DEL)--The DELete key works the same in Number Mode as it does in Command Mode. See "Tour of the Keyboard" in Book 1.

CTRL 4 (CLEAR)--The CLEAR key scrolls the current line and leaves the program line unchanged. The computer then leaves Number Mode. Any changes that were made on the line before you pressed CLEAR are ignored.

CTRL 3 (ERASE)--The ERASE key erases the entire text of the displayed program line. The line number is still displayed.

RESEQUENCE

RESEQUENCE
[initial-line][,increment]
RES

When you enter the RESEQUENCE command, all lines in the program are assigned new line numbers according to the specified initial-line and increment. If no initial-line and increment are specified, the initial-line is 100 and the increment is 10.

The new line number of the first line in the program is the specified initial-line. Succeeding line numbers are assigned using the specified increment.

If you specify only an initial-line, 10 is used as the increment.

If you specify only an increment, 100 is used as the initial-line. Note the comma before the 5 in the example; to specify only an increment, precede the increment with a comma.

All line-number references contained in the program (such as GOTO line-number) are changed to the new line numbers. Any line numbers mentioned in a REM statement are not changed, because they are not essential to the execution of the program.

Both the initial-line and increment must be positive integers.

```
>NEW

>100 A=27.9
>110 B=34.1
>120 PRINT A;B
>RESEQUENCE 20,5
>LIST
 20 A=27.9
 25 B=34.1
 30 PRINT A;B
```

```
>RES 50
>LIST
 50 A=27.9
 60 B=34.1
 70 PRINT A;B
```

```
>NEW

>100 REM THE VALUE OF "A" IS
 PRINTED IN LINE 120
>110 A=A+1
>120 PRINT A
>130 GOTO 110
>RES 10, 5
>LIST
 10 REM THE VALUE OF "A" IS
 PRINTED IN LINE 120
 15 A=A+1
 20 PRINT A
 25 GOTO 15
```

If an invalid line-number reference is used in a program line, the RESEQUENCE command changes the line number reference to 32767. No error message or warning is given.

If you enter a value for the initial-line or increment that creates line numbers greater than 32767, the message BAD LINE NUMBER is displayed. If this error occurs, no line numbers in the program are changed.

If you enter a RESEQUENCE command with no program in memory, the message CAN'T DO THAT is displayed.

```
>NEW

>100 Z=Z+2
>110 PRINT Z
>120 IF Z=50 THEN 150
>130 GOTO 100
>RES 10,5
>LIST
 10 Z=Z+2
 15 PRINT Z
 20 IF Z=50 THEN 32767
 25 GOTO 10
```

```
>RES 32600,100
* BAD LINE NUMBER
>LIST
 10 Z=Z+2
 15 PRINT Z
 20 IF Z=50 THEN 32767
 25 GOTO 10
```

```
>NEW

>RES

 * CAN'T DO THAT
```

SAVE

SAVE CS1
SAVE HEXBUS.device=number.filename

The SAVE command copies the program in the computer's memory to a storage device. The saved program can later be loaded back into the computer's memory with the OLD command.

To save a program to a cassette recorder, position the tape to a blank section, enter SAVE CS1, and the computer displays instructions for you to follow. The screen goes blank during the recording process.

After the program has been copied, you have the option of checking whether your program was recorded correctly. It is recommended that you do so to ensure the accuracy of your tape for future use.

```
 * CHECK TAPE (Y OR N)?
```

If you press N, the cursor appears at the left of the screen. You may then enter NEW to clear the computer's memory. If you press Y, directions for activating the recorder reappear. The screen goes blank during the checking process.

If the check verifies that the data were successfully stored, the message DATA OK is displayed. If an error is detected, an error message appears. You may choose one of these three options:

!o! Press R to record your program again. The same instructions listed previously are displayed.

!o! Press C to repeat the checking procedures. At this point, you may wish to adjust the recorder volume and/or tone controls.

!o! Press E to exit from the recording procedure.

Follow the instructions that appear on the screen. If an additional error message appears indicating that the computer did not properly save your program, follow the displayed instructions, referring to the "Error Messages" section of this book to identify the error involved.

To save a program to a HEX-BUS™ peripheral, you must enter the device=number of the peripheral and the filename to which the program is stored. For example, the statement

```
SAVE HEXBUS.1.MYFILE
```

copies from memory the program stored on peripheral device 1 (the Wafertape™ peripheral) in MYFILE. Refer to the peripheral manuals for the device code for each peripheral and for specific information about filename.

When the SAVE command is performed, the program remains in the memory of the computer, whether or not an error occurred in recording.

```
>SAVE CS1

 * REWIND CASSETTE TAPE
   THEN PRESS ENTER

 * PRESS CASSETTE RECORD
   THEN PRESS ENTER

(The screen goes blank while
the computer records the program.)

 * PRESS CASSETTE STOP
   THEN PRESS ENTER

 * CHECK TAPE (Y OR N)? Y

 * REWIND CASSETTE TAPE
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY
   THEN PRESS ENTER

(The screen goes blank while
the computer checks the program.)

(The computer displays one of three messages.)

 * DATA OK

 * PRESS CASSETTE STOP
   THEN PRESS ENTER


(Or)


 * ERROR - NO DATA FOUND
   PRESS R TO RECORD
   PRESS C TO CHECK
   PRESS E TO EXIT

(Or)


 * ERROR IN DATA DETECTED
   PRESS R TO RECORD
   PRESS C TO CHECK
   PRESS E TO EXIT

 * I/O ERROR 66
```

OLD

OLD CS1
OLD HEXBUS.device=number.filename

The OLD command closes all open files, erases the current program in memory, and reads and loads a previously saved program into the computer's memory. You can then run, list, or edit the program.

To load a program stored on a cassette recorder, enter OLD CS1 and the computer displays instructions for you to follow. The screen goes blank during the reading and loading process.

If the computer does not successfully read your program into memory, the computer prints either ERROR - NO DATA FOUND or ERROR IN DATA DETECTED. You then may choose from these options.

!o! Press R to repeat the reading/loading procedure. Be sure to check the items listed in the XX "Cassette Interface" section in Book 1.

!o! Press C to check that the data in memory and on the tape are the same. At this point, you may wish to adjust the volume and tone controls. Refer to your peripheral manual or to "Tone and Volume Control Settings" in Book 1.

!o! Press E to exit from the procedure.

Follow the instructions that appear on the screen. If an additional error message appears indicating that the computer did not properly load your program into memory, follow the displayed instructions, referring to the "Error Messages" section of this book to identify the error involved.

To load a program that is stored on a HEX-BUS™ peripheral, you must enter the device-number of the peripheral and the filename to which the program is stored. For example, the statement

```
OLD HEXBUS.1.MYFILE
```

loads into memory the program stored on peripheral device 1 (the Wafertape™ peripheral) in MYFILE. Refer to the peripheral manuals for the device code for each peripheral and for specific information about filename.

To execute a program that has been loaded into memory, enter the RUN command when the cursor appears. You can also list the program lines by entering the LIST command.

```
>OLD CS1
 * REWIND CASSETTE TAPE
   THEN PRESS ENTER

 * PRESS CASSETTE PLAY
   THEN PRESS ENTER

(The screen goes blank while
the computer reads the program.)

(The computer displays one of two messages.)

 * DATA OK

 * PRESS CASSETTE STOP CS1
   THEN PRESS ENTER

 (Or)

 * ERROR - NO DATA FOUND
   PRESS R TO READ
   PRESS C TO CHECK
   PRESS E TO EXIT


* I/O ERROR 56
```

DELETE

DELETE "HEXBUS.device=number.filename"

The DELETE command enables you to remove a program or a data file from a mass storage medium on a HEX-BUS peripheral such as the TI Wafertape™ drive. Although the actual file is not deleted, the space it occupies becomes available the next time a file is saved.

The DELETE command must include the device-number of the peripheral and the filename of the file. The filename is a string expression; if a string constant is used, you must enclose it in quotes.

The statement

```
DELETE "HEXBUS.1.DATA"
```

deletes the file stored on device 1 under the name DATA.

You may also remove files stored on some peripheral devices by using the keyword DELETE in the CLOSE statement. The action performed depends on the device used.

```
>500 CLOSE #7:"HEXBUS.1.":DELETE
```

The DELETE command does not delete programs or files stored on audio cassettes.

BREAK

BREAK line-list

The BREAK command is used to set breakpoints in a program to cause the computer to halt or to stop executing the program. When the computer stops at a breakpoint, the message BREAKPOINT AT line-number is displayed. You may then enter any command or any statement that can be used as a command.

BREAK can be entered as a statement in a program with no line-list. When the BREAK statement is encountered, the computer stops running the program.

BREAK can also be entered as a statement in a program with a line-list. BREAK entered as a command must have a line-list. When BREAK is entered with a line-list, breakpoints are set immediately before the lines specified in the line-list. These breakpoints cause the computer to halt before performing each statement in the line-list.

The line-list can be a single line number or a list of line numbers separated by commas.

You can resume program execution (beginning with the line where the breakpoint was set) by entering the CONTINUE command.

A breakpoint does not cause any change in the value of any variables in your program unless you enter a statement that assigns a new value. Note that in this example, C still equals zero because the assignment in statement 110 has not been performed.

You cannot enter the CONTINUE command after you have edited the program (added, deleted, or changed program lines). Otherwise, errors could result from resuming execution of a revised program. If you enter a CONTINUE command after you have edited the program, the message CAN'T CONTINUE is displayed on the screen.

```
>NEW

>100 A=26.7
>110 C=19.3
>115 BREAK
>120 PRINT A+C
>RUN

 * BREAKPOINT AT 115

>100 BREAK 120,130
>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>RUN

 * BREAKPOINT AT 120

>100 BREAK 120,130
>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>BREAK 140
>RUN

 * BREAKPOINT AT 120

>100 BREAK 120,130
>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>RUN

 * BREAKPOINT AT 120
>CONTINUE

 * BREAKPOINT AT 130
>CONTINUE

 60

** DONE **
```

```
>100 BREAK 120,130
>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>RUN

 * BREAKPOINT AT 120

>PRINT X;Y;Z
  10  0  0


>100 BREAK 120
>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>RUN

 * BREAKPOINT AT 120

>110 X=30

>CONTINUE
 * CAN'T CONTINUE
```

When a breakpoint occurs, the breakpoint at that line is removed.

Breakpoints set immediately before program lines can be removed by using the UNBREAK command. These breakpoints are also removed when the line is deleted.

Note that the breakpoint at 110 in the example was removed when the breakpoint occurred, but the breakpoint at 130 was removed by the UNBREAK command.

Breakpoints are removed from all program lines when a SAVE command or a NEW command is entered.

If the line-list specifies a line number less than 1 or greater than 32767, the message BAD LINE NUMBER is displayed and the BREAK command or statement is ignored (no breakpoints are set at any line specified in the line-list).

If the line-list specifies a line number that is a valid line number but is not a line in the program, the message WARNING: BAD LINE NUMBER is displayed. Breakpoints are set at the specified valid line numbers.

```
>110 X=10
>120 Y=20
>130 Z=30
>135 BREAK
>140 PRINT X+Y+Z
>150 GOTO 135
>BREAK 110,120

>RUN

 * BREAKPOINT AT 110
>UNBREAK

>CONTINUE

 * BREAKPOINT AT 135
>CONTINUE
  60

 * BREAKPOINT AT 135


>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>BREAK 110,120130

 * BAD LINE NUMBER


>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z

>BREAK 125,130

 * WARNING:
   BAD LINE NUMBER

>RUN

 * BREAKPOINT AT 130
>CONTINUE
  60

** DONE **
```

UNBREAK XX CHECK TEXT AND EXAMPLES WITH SQUIRREL XX

UNBREAK [line-list]

The UNBREAK command is used to remove breakpoints from program lines that are listed in the line-list of a BREAK command or statement. UNBREAK does not remove the breakpoints that occur when a BREAK statement with no line-list is encountered (for example, 115 BREAK).

The optional line-list following UNBREAK can be a single line number or a list of line numbers, separated by commas, from which you want to remove breakpoints.

The UNBREAK command can also be used as a statement. If an UNBREAK statement is entered with no line-list, all breakpoints are removed. An UNBREAK statement that contains a line-list removes only those breakpoints at the lines specified in that line-list.

```
>NEW

>110 X=10
>115 BREAK
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>BREAK 120,130

>UNBREAK

>RUN

 * BREAKPOINT AT 115

>CONTINUE
  60

 ** DONE **


>NEW

>110 X=10
>120 Y=20
>130 Z=30
>140 PRINT X+Y+Z
>BREAK 120,130,140

>RUN

 * BREAKPOINT AT 120
>UNBREAK 130

>CONTINUE

 * BREAKPOINT AT 140
>CONTINUE
  60

 ** DONE **
```

If the line-list specifies a line number less than one or greater than 32767, the message BAD LINE NUMBER is displayed, and the command is ignored (no breakpoints are removed at any specified line).

If the line-list specifies a line number that is a valid line number but is not a line in the program, the warning BAD LINE NUMBER is displayed. Breakpoints are removed at the valid line numbers specified.

Understanding Subroutines--GOSUB, RETURN, and STOP

A subroutine is a group of lines of programming that performs a specialized routine. Its purpose is to avoid unnecessary duplication of program lines. The lines of a subroutine are written only once in a program, but you can branch to that set of statements as many times as you desire from selected points in the program.

You can branch to a subroutine by means of a GOSUB statement (short for GOto SUBroutine). The GOSUB statement includes the word GOSUB followed by a line number. When a GOSUB statement is encountered, control is transferred to the specified line number. The GOSUB statement is said to "call" the subroutine.

A subroutine can terminate only when it encounters a RETURN statement. Therefore, each subroutine must contain at least one RETURN statement. When a RETURN statement is encountered, control is transferred to the line following the GOSUB statement that called the subroutine.

Subroutines are normally written at the end of a program. A program should have either a STOP statement or some unconditional branching statement immediately before the subroutine(s) so that the computer won't accidentally execute, or "fall into," the subroutines.

When control is transferred to a line within a subroutine, that line and the lines succeeding it are executed.

The program on the right uses a subroutine to print a filler in between printed lines. By using the subroutine to print the filler, you do not have to type lines 230, 240, and 250 each time you wish to print the filler. The filler is printed by simply using a GOSUB statement.

```
>BREAK 130
>UNBREAK 130, 110150

 * WARNING:
   BAD LINE NUMBER

>RUN
  26.7

 * BREAKPOINT AT 130

>CONTINUE
  19.3

 ** DONE **

>BREAK 130

>UNBREAK 130, 105

  * WARNING:
    BAD LINE NUMBER

>RUN
  26.7
  19.3

 ** DONE **
```

CONTINUE

CONTINUE
CON

The CONTINUE command may be entered when the program stops running because of a breakpoint. For an explanation of breakpoints and how they are set, see the BREAK command. Remember that a breakpoint also occurs when BREAK or CLEAR is pressed while the program is running.

You cannot enter the CONTINUE command if you have edited the program (added, deleted, or changed program lines) during a breakpoint. Otherwise, errors could result from starting a revised program in the middle. If you enter a CONTINUE command after you have edited the program, the message CAN'T CONTINUE is displayed on the screen.

```
>NEW

>100 A=9.6
>110 PRINT A
>BREAK 110

>RUN

 * BREAKPOINT AT 110
>CONTINUE
  9.6

 ** DONE **

>BREAK 110

>RUN

 * BREAKPOINT AT 110


>110 A=10.1
>CON
 * CAN'T CONTINUE
```

TRACE

TRACE

The TRACE command enables you to see the order in which the computer performs statements as it runs a program. After you enter the TRACE command, the line number of each program line is displayed before the statement is performed. The TRACE command is most often used to help find errors (such as unwanted infinite loops) in a program.

The TRACE command may also be placed as a statement in a program. The effect of the TRACE command or statement is cancelled when a NEW command or UNTRACE command or statement is performed.

UNTRACE

UNTRACE

The UNTRACE command cancels the effect of the TRACE command. The UNTRACE command may also be used as a statement in a program.

```
>NEW

>100 PRINT "HI"
>110 B=27.9
>120 PRINT :B
>TRACE

>RUN
  <100> HI
  <110><120>
   27.9
 ** DONE **

>UNTRACE

>105 TRACE
>RUN
  HI
  <110> <120>
   27.9

 ** DONE **



>NEW

>100 FOR K=1 TO 2
>110 PRINT K
>120 NEXT K
>TRACE

>RUN
  <100><110> 1
  <120><110> 2
  <120>
 ** DONE **

>UNTRACE

>RUN
  1
  2
 ** DONE **
```

GENERAL PROGRAM STATEMENTS

General program statements do not serve an input-output function. They include the LET statement, which enables you to assign values to variables, the STOP, END, REMark, and program-control statements.

Program control statements, including the GOTO, the ON GOTO, the IF THEN ELSE, the FOR TO STEP, and the NEXT statements enable you to program loops and conditional and unconditional branches.

LET

[LET] variable=expression

The LET statement enables you to assign values to variables in your program. The computer evaluates the expression to the right of the equals sign and puts its value into the variable specified to the left of the equals sign. Note that the keyword LET may be omitted from the assignment statement.

The variable and the expression must correspond in type: numeric expressions must be assigned to numeric variables, and string expressions must be assigned to string variables.

The rules governing overflow and underflow in evaluating a numeric expression apply to the LET statement. (See "Numeric Constants" (XX) for more information.) If the length of an evaluated string expression exceeds 255 characters, the string is truncated on the right, and the program continues. No warning is given.

You may use the relational operators in numeric and string expressions. The result of a relational operator is -1 if the relationship is true and is 0 if the relationship is false.

```
>NEW

>100 LET M=1000
>110 LET C=186000
>120 E=M*C^2
>130 PRINT E
>RUN
  3.4596E+13

 ** DONE **


>NEW

>100 X$="HELLO, "
>110 NAME$="GENIUS!"
>120 PRINT X$;NAME$
>RUN
 HELLO, GENIUS!

 ** DONE **


> NEW

>100 LET A=20
>110 B=10
>120 LET C=A>B
>130 PRINT A;B;C
>140 C=A<B
>150 PRINT A;B;C
>RUN
  20  10 -1
  20  10  0

 ** DONE **
```

REMark

REM remark

The REMark statement enables you to explain and document your program by inserting comments in the program itself. When the computer encounters a REMark statement while running your program, it takes no action but proceeds to the next statement.

You may use any printable character in a REMark statement. The length of the REMark statement is limited by the length of the input line (112 characters or four lines on the screen). If you do not wish to break a word in the middle, press the SPACE BAR repeatedly until the cursor returns to the left side of the screen, and then begin typing again.

END

END

The END statement terminates your program when this statement is executed. END may be used interchangeably with the STOP statement in TI-99/2 BASIC. Although the END statement can appear anywhere in the program, it is normally placed in the last line number of the program to end the program both physically and logically. In contrast, the STOP statement is generally used if you want other termination points in your program.

In TI-99/2 BASIC you are not required to place an END statement in the program. The program automatically stops after it executes the highest-numbered line.

```
> NEW

>100 REM COUNTING FROM 1 TO 1
 0
>110 FOR X=1 TO 10
>120 PRINT X;
>130 NEXT X
>RUN
  1  2  3  4  5  6  7  8  9
  10
 ** DONE **
```

```
>NEW

>100 A=762
>110 B=425
>120 REM NOW PRINT THE SUM OF
  A AND B
>130 PRINT A+B
>RUN
  1187

 ** DONE **
```

```
>NEW

>100 A=10
>110 B=20
>120 C=A*B
>130 PRINT C
>140 END
>RUN
  200

 ** DONE **
```

GOTO

GOTO
    line-number
GO TO

The GOTO statement enables you to transfer control to a specified line within a program. When the computer reaches a GOTO statement, it jumps to the statement specified by the line-number. This is called an unconditional branch.

In the program on the right, line 170 is an unconditional branch; the computer always skips to line 140 at this point. Line 160 is a conditional branch (see IF THEN ELSE); the computer jumps to line 180 only if COUNT and DAYS are equal.

If the specified line-number does not exist in your program, the program stops and prints the message BAD LINE NUMBER.

Note that the space between the words GO and TO is optional.

ON GOTO

ON numeric-expression GOTO
    line-number [,line-number][...]
ON numeric-expression GO TO

The ON GOTO statement tells the computer to jump to one of several program lines, depending on the value of the numeric-expression.

The computer first evaluates the numeric-expression and rounds the result to an integer. This integer becomes a pointer for the computer, indicating which program line in the ON GOTO statement to perform next. If the value of the numeric-expression is 1, the computer proceeds to the statement specified by the first line-number. If the value is 2, the computer proceeds to the statement specified by the second line-number, and so on.

If the rounded value of the numeric-expression is less than 1 or greater than the number of line-numbers listed in the ON GOTO statement, the program stops and prints BAD VALUE IN line-number. If the line-number you specify is outside the range of line numbers in your program, the message BAD LINE NUMBER is displayed and the program stops running.

```
>100 K=10
>110 PRINT "K= ";K
>120 K=K*2
>130 GOTO 110
>RUN
  10
  20
  30
   .
   .
   .
(Press BREAK to stop the program.)
```

```
>NEW

>100 INPUT X
>110 ON X GOTO 120,140,160,18
 0,200
>120 PRINT "X=1"
>130 GOTO 100
>140 PRINT "X=2"
>150 GOTO 100
>160 PRINT "X=3"
>170 GOTO 100
>180 PRINT "X=4"
>190 GOTO 100
>200 END
>RUN
 ? 2
 X=2
 ? 1.2
 X=1
 ? 3.7
 X=4
 ? 6

 * BAD VALUE IN 110
```

IF THEN ELSE

IF condition THEN line1 [ELSE line2]

The IF THEN ELSE statement enables you to change the normal sequence of program execution by using a conditional branch.

The computer evaluates the condition included in the statement as either true or false.

If the condition is true, the computer jumps to line1, the line number following the word THEN.

If the condition is false and the ELSE option is used, the computer jumps to line2, the line number following the word ELSE.

If the condition is false and ELSE is omitted, the computer continues with the next program line.

The condition being tested can be a relational expression or a numeric expression. Relational expressions evaluate to either true or false. Numeric expressions evaluate to 0 or nonzero values; only a zero value is considered false.

In relational expressions, numeric expressions must be compared to numeric expressions and string-expressions to string expressions.

Numeric-expressions are compared algebraically.

String-expressions are compared left-to-right, character by character, using the ASCII character codes. A character with a lower ASCII code is considered less than one with a higher ASCII code. Thus, you can sort strings into numeric or alphabetic order. If one string is longer than the other, the comparison is made for each character in the shorter string. If there is no difference, the computer considers the longer string to be greater.

The condition can be a logical expression by using multiplication for a logical AND and addition for a logical OR. The condition in the example,

```
IF (5<3)*(3<4) THEN 150          [IF (5<3)AND(3<4) THEN 150]
```

is false. The product of (5<3)*(3<4) is zero (false) because one of its factors is zero (false).

The condition in the example,

```
IF (5<3)+(3<4) THEN 150          [IF (5<3)OR(3<4) THEN 150]
```

is true (has a nonzero value) because even though 5<3 is false and has a value of zero (0), 3<4 is true and has a nonzero value, which when added to zero produces a nonzero result.

```
>NEW

>100 INPUT "HOW MANY VALUES?":N
>110 INPUT "VALUE?":A
>120 L=A
>130 N=N-1
>140 IF N<=0 THEN 170
>150 INPUT "VALUE?":A
>160 IF L>A THEN 130 ELSE 120
>170 PRINT L;"IS THE LARGEST"
>RUN
 HOW MANY VALUES?3
 VALUE?456
 VALUE?321
 VALUE?292
  456 IS THE LARGEST

 ** DONE **

>NEW

>100 INPUT "A$ IS ":A$
>110 INPUT "B$ IS ":B$
>120 IF A$=B$ THEN 160
>130 IF A$<B$ THEN 180
>140 PRINT "B$ IS LESS"
>150 GOTO 190
>160 PRINT "A$=B$"
>170 GOTO 190
>180 PRINT "B$ IS GREATER"
>190 END
>RUN
 A$ IS TEXAS
 B$ IS TEX
 B$ IS LESS

 ** DONE **

>RUN
 A$ IS TAXES
 B$ IS TEX
 B$ IS GREATER

 ** DONE **

>NEW

>100 INPUT "A IS ":A
>110 INPUT "B IS ":B
>120 IF A+B THEN 150
>130 PRINT "RESULT IS ZERO, E
 XPRESSION FALSE"
>140 GOTO 100
>150 PRINT "RESULT IS NON-ZER
 O, EXPRESSION TRUE"
>160 GOTO 100
>RUN
 A IS 2
 B IS 3
 RESULT IS NON-ZERO, EXPRESSION TRUE
 A IS 2
 B IS -2
```

(Press BREAK to end loop.)

FOR TO STEP

FOR control-variable=initial-value TO limit [STEP increment]

The FOR TO STEP statement and the NEXT statement are used together to form a FOR-NEXT loop, which may be used for programming repetitive processes.

The values you assign to the initial-value, limit, and increment determine how many times the loop is repeated. The control-variable is a numeric variable that acts as a counter for the loop. When the FOR TO STEP statement is performed, the control-variable is set to the initial-value. The computer then performs program statements until it encounters a NEXT statement.

When the NEXT statement is encountered, the computer adds the optional STEP increment to the control-variable. If STEP is omitted, the computer uses an increment of +1. (If the increment is a negative value, the control-variable is reduced by the STEP amount.) The computer then compares the control-variable to the value of the limit. If the control-variable does not yet exceed the limit, the computer repeats the statements following the FOR TO STEP statement until the NEXT statement is again encountered. If the new value for the control-variable is greater than the limit (or less, if the increment is negative), the computer leaves the loop and continues with the program statement following the NEXT statement. The value of the control-variable is not changed when the computer leaves the FOR-NEXT loop.

The limit and the STEP increment are numeric expressions that are evaluated once during a loop performance (when the FOR TO STEP statement is first encountered) and remain in effect until the loop is finished. Any changes made in these values while a loop is in progress have no effect on the number of times the loop is performed.

```
>NEW

>100 REM COMPUTING SIMPLE INT
 EREST FOR 10 YEARS
>110 INPUT "PRINCIPLE? ":P
>120 INPUT "RATE? ":R
>130 FOR YEARS=1 TO 10
>140 P=P+(P*R)
>150 NEXT YEARS
>160 P=INT(P*100+.5)/100
>170 PRINT P
>RUN
 PRINCIPLE? 100
 RATE? .0775
  210.95

 ** DONE **


>NEW

>100 REM EXAMPLE OF FRACTIONA
 L INCREMENT
>110 FOR X=.1 TO 1 STEP .2
>120 PRINT X;
>130 NEXT X
>140 PRINT :X
>RUN
   .1  .3  .5  .7  .9

   1.1

 ** DONE **


>NEW

>100 L=5
>110 FOR K=1 TO L
>120 L=20
>130 PRINT L;K
>140 NEXT K
>RUN
   20  1
   20  2
   20  3
   20  4
   20  5

 ** DONE **
```

If you change the value of the control-variable while the loop is being performed, the number of times the loop is repeated is changed.

In TI-99/2 BASIC the expressions for initial-value, limit, and increment are evaluated before the initial-value is assigned to the control-variable. Thus, in line 110 of the program on the right, the value 5 is assigned to the limit before the control-variable K is assigned a value. The loop is repeated five times, not just once.

The sign of the control-variable can change during the performance of a FOR-NEXT loop.

If the initial-value is greater than the limit (or less than the limit for a negative increment), the loop is skipped and the program continues with the statement following the NEXT statement.

If the value of the increment is zero, the computer displays the error message BAD VALUE IN line-number and the program stops running.

After you enter a RUN command, but before your program is performed, the computer verifies that you have the same number of FOR TO STEP and NEXT statements. If you do not have the same number, the message FOR-NEXT ERROR is displayed and the program is not run.

```
>NEW

>100 FOR K=1 TO 10
>110 K=K+1
>120 PRINT K
>130 NEXT K
>140 PRINT K
>RUN
  2
  4
  6
  8
  10
  11

 ** DONE **

>NEW

>100 M=5
>110 FOR K=1 TO M
>120 PRINT K;
>130 NEXT K
>RUN
  1  2  3  4  5

 ** DONE **


>NEW

>100 FOR K=2 TO -3 STEP -1
>110 PRINT K;
>120 NEXT K
>RUN
  2  1  0 -1 -2 -3

 ** DONE **

>NEW

>100 REM INITIAL VALUE TOO GR
 EAT
>110 FOR K=6 TO 5
>120 PRINT K
>130 NEXT K
>RUN

 ** DONE **
```

FOR TO STEP

FOR-NEXT loops may be "nested"; that is, one FOR-NEXT loop may be contained wholly within another. Be careful, however, to observe the following conventions:

Each FOR TO STEP statement must be paired with a NEXT statement.

Different control-variables must be used for each nested FOR-NEXT loop.

If a FOR-NEXT loop contains any portion of another FOR-NEXT loop, it must contain all of that FOR-NEXT loop. Otherwise, the computer stops running the program and prints the error message CAN'T DO THAT IN line-number.

You may branch out of a FOR-NEXT loop using GOTO, ON GOTO, or IF THEN ELSE statements, but you may not branch into a FOR-NEXT loop using these statements. You may, however, use GOSUB or ON GOSUB statements to leave a FOR-NEXT loop and then return to the loop. Be sure that you do not use the same control-variable for any FOR-NEXT loops you may have in subroutines.

```
>NEW

>100 REM FIND THE LOWEST THRE
 E DIGIT NUMBER EQUAL TO THE
 SUM OF THE CUBES OF ITS DIGI
 TS
>110 FOR HUND=1 TO 9
>120 FOR TENS=0 TO 9
>130 FOR UNITS=0 TO 9
>140 SUM=100*HUND+10*TENS+UNI
 TS
>150 IF SUM<>HUND^3+TENS^3+UN
 ITS^3 THEN 180
>160 PRINT SUM
>170 GO TO 210
>180 NEXT UNITS
>190 NEXT TENS
>200 NEXT HUND
>210 END
>RUN
  153

 ** DONE **
```

```
>NEW

>100 FOR K=1 TO 3
>110 PRINT K
>120 GOSUB 140
>130 NEXT K
>140 FOR K=1 TO 5
>150 PRINT K;
>160 NEXT K
>170 RETURN
>RUN
  1
  1  2  3  4  5
 * CAN'T DO THAT IN 130
```

NEXT

NEXT control-variable

The NEXT statement is always paired with the FOR TO STEP statement. The control-variable is the same one that appears in the corresponding FOR TO STEP statement.

The NEXT statement actually controls whether the computer repeats the loop or exits to the program line following the NEXT statement. When a NEXT statement is performed, the computer adds the previously evaluated increment in the STEP to the control-variable and then tests the control-variable to see if it exceeds the previously evaluated limit specified in the FOR TO STEP statement. If the control-variable does not exceed the limit, the loop is repeated.

```
>NEW

>100 FOR X=1 TO 10
>110 PRINT X;
>120 NEXT X
>RUN
  1  2  3  4  5  6  7  8  9  10

 ** DONE **
```

```
>NEW

>100 REM ROCKET COUNTDOWN
>110 CALL CLEAR
>120 FOR K=10 TO 1 STEP -1
>130 PRINT K
>140 FOR DELAY=1 TO 400
>150 NEXT DELAY
>160 CALL CLEAR
>170 NEXT K
>180 PRINT "BLAST OFF!"
>RUN

(Computer flashes countdown.)

BLAST OFF!

** DONE **
```

INPUT-OUTPUT STATEMENTS

INPUT-OUTPUT statements (PRINT, DISPLAY, INPUT, READ, DATA, RESTORE) enable you to transfer data in and out of your program.

Data can be input to your program from three types of sources:

- from the keyboard--using the INPUT statement
- internally from the program itself--using the READ, DATA, and RESTORE statements
- from files stored on peripheral devices--using the INPUT statement

Data can be output to two types of devices:

- the screen--using the PRINT and DISPLAY statements
- files stored on peripheral devices--using the PRINT statement

Refer to the "File Processing" section of this manual for information on using input-output statements with peripheral devices.

INPUT

INPUT [input-prompt:] variable-list

(For information on the use of the INPUT statement with a file, see the "File Processing" section.)

This form of the INPUT statement is used when you enter data via the keyboard. The INPUT statement causes the program to pause until valid data are entered.

Although the computer usually accepts up to one input line (4 lines on your screen) for each INPUT statement, a long list of values may be rejected by the computer. If you receive the message LINE TOO LONG after entering an input line, divide the lengthy line into at least two separate INPUT statements.

Entering the Input Statement

When an input-prompt is used, it must be followed by a colon. The input-prompt is a string expression (constant or variable) that can be used to prompt for values to be entered from the keyboard (if a string constant is used, it must be enclosed in quotation marks). The INPUT statement displays the input-prompt message and waits for data to be entered.

When an input-prompt is not used, the computer displays a question mark (?) followed by a space and waits for data to be entered.

The variable-list contains one or more variables that are assigned values when the INPUT statement is performed. The variables may be numeric and/or string variables. If the variable-list contains two or more variables, they must be separated by commas. The values to be assigned to these variable names must also be separated by commas.

```
>NEW

>100 INPUT B
>110 PRINT B
>RUN
 ? 25
  25

 ** DONE **
```

```
>NEW
>100 INPUT "COST OF CAR: ":B
>110 A$="TAX: "
>120 INPUT A$:C
>130 INPUT "SALES "&A$:X
>140 PRINT B;C;X
>RUN
 COST OF CAR: 5500
 TAX: 500
 SALES TAX: 500
  5500  500  500

 ** DONE **
```

```
>NEW

>100 INPUT A,B$,C,D
>110 PRINT A:B$:C:D
>RUN
 ? 10,HELLO,25,3.2
  10
 HELLO
  25
  3.2

 ** DONE **
```

INPUT

Responding to an Input Statement

When an INPUT statement with more than one variable in the variable-list is performed, the values corresponding to the variables must be entered in the same order as they are listed in the INPUT statement. All the values must be entered in one input line (up to 4 screen lines) and must be separated by commas. When entering string values, you may enclose the string in quotes, although the quotation marks are not required. However, a string that contains a comma, a quotation mark, or leading or trailing spaces must be enclosed in quotes.

Variables are assigned values from left to right in the variable-list. Thus, subscript expressions in the variable-list are not evaluated until variables to the left have been assigned values.

```
>NEW

>100 INPUT A$
>110 PRINT A$::
>120 INPUT B$
>130 PRINT B$::
>140 INPUT C$
>150 PRINT C$::
>160 INPUT D$
>170 X=500
>180 PRINT D$;X::
>RUN
 ? "JONES, MARY"
JONES, MARY

? """HELLO THERE"""
"HELLO THERE"

? "JAMES B. SMITH, JR."
JAMES B. SMITH, JR.

? "SELLING PRICE IS "
SELLING PRICE IS  500

** DONE **


>NEW

>100 INPUT K,A(K)
>110 PRINT K:A(K)
>RUN
 ? 3,7
  3
  7

 ** DONE **
```

INPUT

When you enter information in response to an INPUT statement, the information is validated by the computer. If the input data are invalid, the message

```
* WARNING:
  INPUT ERROR IN line-number
TRY AGAIN:
```

appears on the screen, and you must reenter the data. The computer determines the following input to be invalid:

!o! Data that contain more or fewer values than requested by the INPUT statement.

!o! A string constant entered when a number is required. (Note: A number is a valid string, so you may enter a number when a string constant is required.)

If you enter a number that causes an overflow, the message

```
* WARNING:
  NUMBER TOO BIG IN line-number
TRY AGAIN:
```

appears on the screen and you must reenter the data. If you enter a number that causes an underflow, the value is replaced by zero. No warning message is given.

```
>NEW

>100 INPUT A,B$
>110 PRINT A;B$
>RUN
 ? 12,HI,3

 * WARNING:
   INPUT ERROR IN 100
 TRY AGAIN: HI,3

 * WARNING:
   INPUT ERROR IN 100
 TRY AGAIN: 23,HI
  23 HI

 ** DONE **


>NEW

>100 INPUT A
>110 PRINT A
>RUN
 ? 23E139

 * WARNING:
   NUMBER TOO BIG IN 100
 TRY AGAIN: 23E-139
  0

 ** DONE **
```

READ

READ variable-list

The READ statement enables you to read data stored inside your program in DATA statements. The variable-list specifies those variables that are to be assigned values. Variable names in the variable-list may include numeric variables and/or string variables.

The computer reads each DATA statement sequentially from left to right and assigns values to the variables in the variable-list from left to right. Subscript expressions in the variable-list are not evaluated until variables to the left have been assigned.

Each time a READ statement is performed, the variables in its variable-list are assigned values from a DATA statement. If a DATA statement does not contain enough values to assign to the variables, the READ statement assigns the values in the next DATA statement until all the variables have been assigned a value. If a READ statement does not assign all the values in a DATA statement, the next READ statement performed assigns the next unread data value(s).

DATA statements are normally read in line-number order. You can override this sequencing, however, by using the RESTORE statement.

By following the program on the right, you can see how the READ, DATA, and RESTORE statements interact. In line 120, the computer begins assigning values to A and B from the DATA statement with the lowest line number, line 180. The first READ, therefore, assigns A=2 and B=4. The next performance of the READ statement still takes data from line 180 and assigns A=6, B=8. The third READ statement assigns the last item in line 180 to the variable A and the first item in line 190 to the variable B, so that A=10 and B=12. The fourth READ, the last in the J-loop, continues to get data from line 190, so that A=14 and B=16. Before going through the K-loop again, however, the computer encounters a RESTORE statement in line 160, which directs it to get data from the beginning of line 190 for the next READ statement. The computer then completes the program by reading the data from line 190 and then from line 200.

```
>NEW

>100 FOR K=1 TO 3
>110 READ X,Y
>120 PRINT X;Y
>130 NEXT K
>140 DATA 22,15,36,52,48,96.5

>RUN
  22  15
  36  52
  48  96.5

 ** DONE **


>NEW

>100 READ K,A(K)
>110 DATA 2,35
>120 PRINT A(K)
>RUN
  35

 ** DONE **


>NEW

>100 FOR K=1 TO 2
>110 FOR J=1 TO 4
>120 READ A,B
>130 PRINT A;B;
>140 NEXT J
>150 PRINT
>160 RESTORE 190
>170 NEXT K
>180 DATA 2,4,6,8,10
>190 DATA 12,14,16,18
>200 DATA 20,22,24,26
>RUN
  2  4  6  8  10  12  14  16

  12  14  16  18  20  22  24

 26

** DONE **
```

READ

When data are read from a DATA statement, the type of data in the data list and the type of variables to which the values are assigned must correspond. If you try to assign a string value to a numeric variable, the message DATA ERROR IN line-number of the READ statement where the error occurs appears on the screen, and the program stops running. Remember that a number is a valid string, so numbers may be assigned to either string or numeric variables.

When a READ statement is performed, if there are more names in the variable-list than values remaining in DATA statements, a DATA ERROR message is displayed on the screen and the program stops running.

If a numeric constant that causes an underflow is read, its value is replaced by zero--no warning is given--and the program continues running normally. If a numeric constant that causes an overflow is read, its value is replaced by the appropriate computer limit, the message WARNING: NUMBER TOO BIG IN line-number is displayed on the screen, and the program continues. For information on underflow, overflow, and numeric limits, see "Numeric Constants."

```
>NEW

>100 READ A,B
>110 DATA 12,HELLO
>120 PRINT A;B
>RUN

 * DATA ERROR IN 100



>NEW

>100 READ A,B
>110 DATA 12E-135
>120 DATA 36E142
>130 PRINT :A:B
>140 READ C
>RUN

 * WARNING:
   NUMBER TOO BIG IN 100

  0
  9.99999E+**

  * DATA ERROR IN 140
```

DATA

DATA data-list

The DATA statement enables you to store data within your program. The data-list contains values assigned to the variables specified in the variable list of a READ statement. The values are assigned when the READ statement is performed. Items in the data-list are separated by commas. When the computer encounters a DATA statement, it proceeds to the next statement with no other effect.

DATA statements may appear anywhere in a program, but the order in which they appear is important. Data from the data-lists are read sequentially, beginning with the first item in the first DATA statement. If your program includes more than one DATA statement, the DATA statements are read in ascending line-number order unless otherwise specified by a RESTORE statement. Thus, the order in which the data appear within the data-list and the order of the DATA statements within the program normally determine the order in which the data are read.

Each value in the data-list must correspond to the type of the variable to which it is assigned. Thus, if a numeric variable is specified in the READ statement, a numeric constant must be in the corresponding place in the DATA statement. Remember that a number is a valid string, so you may have a number in the corresponding place in the DATA statement when a string constant is required.

In a DATA statement, string constants that contain a comma, a quotation mark, or leading or trailing spaces must be enclosed in quotation marks. If a string constant does not contain one of these characters, you may omit the quotation marks.

If a DATA statement contains adjacent commas, the computer assigns a null string (a string with no characters) to the variable being assigned. In the example on the right, the DATA statement in line 110 contains two adjacent commas. Thus, a null string is assigned to B$.

```
>NEW

>100 FOR K=1 TO 5
>110 READ A,B
>120 PRINT A;B
>130 NEXT K
>140 DATA 2,4,6,7,8
>150 DATA 1,2,3,4,5
>RUN
  2  4
  6  7
  8  1
  2  3
  4  5

 ** DONE **

>NEW

>100 READ A$,B$,C,D
>110 PRINT A$:B$:C:D
>120 DATA HELLO,"JONES, MARY"
 ,28,3.1416
>RUN
HELLO
JONES, MARY
 28
 3.1416

 ** DONE **


>NEW

>100 READ A$,B$,C
>110 DATA HI,,2
>120 PRINT "A$ IS ";A$
>130 PRINT "B$ IS ";B$
>140 PRINT "C IS ";C
>RUN
A$ IS HI
B$ IS
C IS  2

** DONE **
```

RESTORE

RESTORE [line-number]

(See the "File Processing" section for information about using RESTORE in file processing.)

This form of the RESTORE statement tells your program which DATA statement to use with the next READ statement.

When RESTORE is used with no line-number, the next READ statement performed assigns values beginning with the first value in the first DATA statement in the program.

When RESTORE is followed by the line-number of a DATA statement, the next READ statement performed assigns values beginning with the first value in the DATA statement specified by the line-number.

If the line-number specified in a RESTORE statement is neither a DATA statement nor a program line-number, the next READ statement performed starts at the first DATA statement whose line-number is greater than the one specified. If there is no DATA statement with a line-number greater than or equal to the one specified and a READ statement is performed, the error message DATA ERROR is displayed. If the line-number specified is greater than the highest line-number in the program, the program stops running and the message DATA ERROR IN line-number is displayed.

```
>100 FOR K=1 TO 2
>110 FOR J=1 TO 4
>120 READ A
>130 PRINT A;
>140 NEXT J
>150 RESTORE 180
>160 NEXT K
>170 DATA 12,33,41,26,42,50
>180 DATA 10,20,30,40,50
>RUN
  12  33  41  26  10  20  30
  40

 ** DONE **

>NEW

>100 FOR K=1 TO 5
>110 READ X
>120 RESTORE
>130 PRINT X;
>140 NEXT K
>150 DATA 10,20,30
>RUN
  10  10  10  10  10

 ** DONE **

>NEW

>100 READ A,B
>110 RESTORE 130
>120 PRINT A;B
>130 READ C,D
>140 PRINT C;D
>150 DATA 26.9,34.67
>RUN
  26.9  34.67
  26.9  34.67

 ** DONE **

>110 RESTORE 145
>RUN
  26.9  34.67
  26.9  34.67

 ** DONE **

>110 RESTORE 155
>RUN
  26.9  34.67

 * DATA ERROR IN 110
```

DISPLAY

DISPLAY [print-list]

The DISPLAY statement is identical to the PRINT statement when you use it to print items on the screen. The DISPLAY statement cannot be used to write to any device except the screen. For a complete discussion of how to use this statement, see the instructions for the PRINT statement.

```
>NEW

>100 A =35.6
>110 B$="HI!!"
>120 C=49.7
>130 PRINT B$:A;C
>140 DISPLAY B$:A;C
>RUN
 HI!!
 35.6  49.7
HI!!
 35.6  49.7

** DONE **
```

PRINT

PRINT [print-list]

(For information on using the PRINT statement with files, see the "File Processing" section.)

The PRINT statement lets you print numbers and strings on the screen. The print-list consists of

- print items--numeric expressions and string expressions to be printed on the screen and TAB functions that control print positioning (similar to the TAB key on the typewriter).
- print separators--the punctuation (commas, semicolons, and colons) between print items serving to indicate the positioning of the data on the print line.

When the computer performs a PRINT statement, the values of the expressions in the print-list are displayed on the screen in order from left to right, as specified by the print separators and TAB functions.

Printing Strings

String expressions in the print-list are evaluated to produce a string result. There are no blank spaces inserted before or after a string. If you wish to print a blank space before or after a string, you must include the space in the string or insert it separately within quotation marks.

Printing Numbers

Numeric expressions in the print-list are evaluated to produce a numeric result. Positive numbers are printed with a leading space (instead of a plus sign), and negative numbers are printed with a leading minus sign. All numeric values are printed with a trailing space.

```
>NEW

>100 A=10
>110 B=20
>120 STRING$="TI COMPUTER"
>130 PRINT A;B:STRING$
>140 PRINT "HELLO, FRIEND"
>RUN
 10  20
 TI COMPUTER
 HELLO FRIEND

 ** DONE **


>NEW

>100 N$="JOAN"
>110 M$="HI"
>120 PRINT M$;N$
>130 PRINT M$&" "&N$
>140 PRINT "HELLO ";N$
>RUN
HIJOAN
HI JOAN
HELLO JOAN

** DONE **


>NEW

>100 LET A=10.2
>110 B=-30.5
>120 C=16.7
>130 PRINT A;B;C
>140 PRINT A+B
>RUN
 10.2 -30.5  16.7
-20.3

** DONE **
```

PRINT

The PRINT statement displays numbers in either normal decimal form or scientific notation, according to these rules:

1. All numbers with 10 or fewer digits are printed in normal decimal form.

2. Integer numbers with more than 10 digits are printed in scientific notation.

3. Non-integer numbers with more than 10 digits are printed in scientific notation only if they can be presented with more significant digits in scientific notation than in normal decimal form. If printed in normal decimal form, all digits beyond the tenth digit are omitted.

If numbers are printed in normal decimal form, the following conventions are observed:

- Integers are printed without decimal points.

- Non-integers are printed with decimal points in proper position. Trailing zeros after the decimal point are omitted. If the number has more than 10 digits, it is rounded to 10 digits.

- A 0 (zero) is not printed by itself to the left of the decimal.

If numbers are printed in scientific notation, the format is:

mantissa E exponent

and the following rules apply:

- The mantissa is printed with six or fewer digits, with one digit to the left of the decimal point.

- Trailing zeros are omitted after the decimal point of the mantissa.

- If there are more than five digits after the decimal point in the mantissa, the fifth digit is rounded.

- The exponent is a two-digit number displayed with a plus or minus sign.

- If you attempt to print a number with an exponent greater than +99 or less than -99, the computer prints ** following the sign of the exponent.

```
>PRINT -10;7.1
 -10  7.1

>PRINT 93427685127
  9.34277E+10

>PRINT 1E-10
  .0000000001

>PRINT 1.2E-10
  1.2E-10

>PRINT .000000000246
  2.46E-10


>PRINT 15;-3
  15 -3

>PRINT 3.350;-46.1
  3.35 -46.1

>PRINT 791.123456789
  791.1234568

>PRINT -12.7E-3;0.64
 -.0127  .64


>PRINT .0000000001978531
  1.97853E-10

>PRINT -98.77E21
 -9.877E+22

>PRINT 736.400E10
  7.364E+12

>PRINT 12.36587E-15
  1.23659E-14

>PRINT 1.25E-9;-43.6E12
 1.25E-09 -4.36E+13

>PRINT .76E126;81E-115
  7.6E+**  8.1E-**
```

PRINT

Print Separators

Each screen line used with the PRINT statement has 28 character positions numbered from left to right (1 through 28). Each line is divided into two 14-character print zones. By using the print separators and the TAB function, you can control the position of the print items displayed on the screen.

There are three types of print separators: semicolons (;), colons (:), and commas (,). At least one print separator must be placed between adjacent print items in the print-list. Multiple print separators may be used side by side and are evaluated from left to right.

The semicolon print separator (;) causes the next print item to print immediately after the previous item printed, with no extra spaces. In the program on the right, the spaces after the numbers appear only because all numbers are printed with a trailing space regardless of the type of print separator used.

The colon print separator (:) causes the next print item to print at the beginning of the next line. Each extra colon causes one blank line to appear; the colon's function is similar to that of a typewriter carriage return.

The comma print separator (,) causes the next print item to print at the beginning of the next print zone. Print lines are divided into two zones. The first zone begins in column 1 on the screen and the second begins in column 15. If the first print zone is already full when a comma print separator is evaluated, the next print item begins on the next line.

```
>NEW

>100 A=-26
>110 B=-33
>120 C$="HELLO"
>130 PRINT A;B;C$
>RUN
 -26 -33 HELLO

 ** DONE **


>PRINT "A"::"B"
 A

 B


>NEW

>100 A=-26
>110 B$="HELLO"
>120 PRINT A:B$
>RUN
 -26
 HELLO

 ** DONE **


>NEW

>100 A$="ZONE 1"
>110 B$="ZONE 2"
>120 PRINT A$,B$
>130 PRINT A$:,B$,A$
>RUN
 ZONE 1         ZONE 2
 ZONE 1
                ZONE 2
 ZONE 1

 ** DONE **
```

PRINT

TAB Function

The TAB function specifies the starting position on the print line for the next print item. Note that the TAB function cannot be used with INTERNAL type files. The format of the TAB function is:

TAB(numeric-expression)

The numeric-expression is evaluated and rounded to the nearest integer D. If D is less than 1, its value is replaced by 1. If D is greater than 28, D is repeatedly reduced by 28 until $1 \leq D \leq 28$. If the number of characters already printed on the current line is less than or equal to D, the next print item is printed on the same line beginning in position D. If the number of characters already printed on the current line is greater than D, the next item is printed on the next line beginning in position D.

Note that the TAB function is a print item and thus must be preceded by a print separator, except when it is the first item in the print-list. The print separator before a TAB function is evaluated before the TAB function, and the print separator following the TAB function is evaluated after the TAB function.

In the program on the right, the computer does the following:

!o! line 120--prints A, moves to column 17, prints B.
!o! line 130--prints A, moves to the next print zone, prints B.
!o! line 140--prints A, moves to column 20, moves to the next print zone because of the comma (in this case column 1 of the next screen line), prints B.
!o! line 150--moves to column 5, prints A, moves to column 6 of the next line (because column 6 of the current line was passed when A was printed), prints B.
!o! line 160--prints A, subtracts 28 from 43 to begin the TAB function within the allowable character positions, moves to position 15 (43-28=15), prints B.

```
>NEW

>100 A=23.5
>110 B=48.6
>120 MSG$="HELLO"
>130 PRINT TAB(5);MSG$;TAB(33
 );MSG$
>140 PRINT A;TAB(10);B
>150 PRINT TAB(3);A;TAB(3);B
>RUN
     HELLO
     HELLO
  23.5      48.6
    23.5
    48.6

 ** DONE **


>NEW

>100 A=326
>110 B=79
>120 PRINT A;TAB(17);B
>130 PRINT A,B
>140 PRINT A;TAB(20),B
>150 PRINT TAB(5);A;TAB(6);B
>160 PRINT A;TAB(43);B
>RUN
  326             79
  326             79
  326
  79
       326
        79
  326             79

 ** DONE **
```

PRINT

A print item following a TAB is not split between two screen lines unless the print item is a string with more than twenty-eight characters. In that case, the string always begins on a new line. If a numeric print item can be printed on the current line without its trailing space, the number is printed on the current line. If the entire number itself will not fit on the current line, it is printed on the next line.

The print-list may end with a print separator. If it does, the print separator is evaluated and the first print item in the next PRINT statement (line 160) starts in the position indicated by the print separator.

If the print-list is not terminated by a print separator (line 130), the computer considers the current line complete when all the print items are printed. In this case the first print item in the next PRINT statement (line 140) always begins on a new line.

You may use a PRINT statement with no print-list. When such a PRINT statement is performed, the computer advances to the first character position of the next screen line. This has the effect of skipping a line if the preceding PRINT statement does not end with a print separator.

```
>NEW

>100 A=23767
>110 B=79856
>120 C=A+B
>130 D=B-A
>140 PRINT A;B;C;D
>150 PRINT "A=";A;"B=";B;"C="
 ;C;"D=";D
>RUN
  23767  79856  103623  56089
 A= 23767 B= 79856 C= 103623
 D= 56089

 ** DONE **

>NEW

>100 A=23
>110 B=597
>120 PRINT A,
>130 PRINT B
>140 PRINT A;B;
>150 C=468
>160 PRINT C
>RUN
  23             597
  23  597  468

 ** DONE **

>NEW

>100 A=20
>110 PRINT A
>120 PRINT
>130 B=15
>140 PRINT B
>RUN
  20

  15

 ** DONE **
```

BUILT-IN NUMERIC FUNCTIONS

The numeric functions described in this section are built into TI-99/2 BASIC and perform some of the frequently used arithmetic operations. These functions eliminate a large amount of programming necessary to obtain equivalent results.

The built-in functions that are used with strings are discussed in the "Built-in String Functions" section.

ABS--Absolute Value

ABS(numeric-expression)

The ABSolute value function returns the absolute value of the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

If the argument is positive, ABS returns the argument itself. If the argument is negative, ABS returns the negative of the argument. Thus, for the argument X,

!o! If X>=0, ABS(X)=X

!o! If X<0, ABS(X)=-X
(i.e., ABS(-3)=-(-3)=3

ATN--Arctangent

ATN(numeric-expression)

The arctangent function (ATN) returns the arctangent of the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

ATN(X) returns the angle (in radians) whose tangent is X. To express the angle in degrees, multiply the answer by (180/(4*ATN(1))) or 57.295779513079, which is 180/pi.

The value given for ATN is always between -pi/2 and pi/2.

```
>NEW

>100 A=-27.36
>110 B=9.7
>120 PRINT ABS(A);ABS(B)
>130 PRINT ABS(3.8);ABS(-4.5)

>140 PRINT ABS(-3*2)
>150 PRINT ABS(A*(B-3.2))
>RUN
  27.36  9.7
  3.8  4.5
  6
  177.84

 ** DONE **
```

```
>NEW

>100 PRINT ATN(.44)
>110 PRINT ATN(1E127)
>120 PRINT ATN(1E-129);ATN(0)

>130 PRINT ATN(.3)*57.2957795
 13079
>140 PRINT ATN(.3)*(180/(4*AT
 N(1)))
>RUN
  .4145068746
  1.570796327
  0  0
  16.69924423
  16.69924423

** DONE **
```

COS--Cosine

COS(numeric-expression)

The COSine function returns the cosine of the argument X, where X is an angle in radians. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

If the angle is in degrees, multiply the degrees by pi/180 to find the equivalent angle in radians. You may use (4*ATN(1))/180 or 0.0174532925199 4 for pi/180. Note that if you enter a value of X where
$|X| \geq 1.570796326637 5 \times 10^{10}$, the message BAD ARGUMENT is displayed and the program stops running.

EXP--Exponential

EXP(numeric-expression)

The EXPonential function returns the value of e raised to the power of the argument X (e^{X}, where e=2.718281828). The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

EXP is the inverse of the natural logarithm function (LOG). Thus, X=EXP(LOG(X)).

```
>NEW

>100 A=1.047197551196
>110 B=60
>120 C=.01745329251994
>130 PRINT COS(A);COS(B*C)
>140 PRINT COS(B*(4*ATN(1))/1
 80)
>RUN
  .5  .5
  .5

 ** DONE **

>PRINT COS(2.2E11)

 * BAD ARGUMENT
```

```
>NEW

>100 A=3.79
>110 PRINT EXP(A);EXP(9)
>120 PRINT EXP(A*2)
>130 PRINT EXP(LOG(2))
>RUN
  44.25640028  8103.083928
  1958.628965
  2

 ** DONE **
```

INT--Integer

INT(numeric-expression)

The INTeger function returns the largest integer that is not greater than the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

If you specify an integer as the argument, the same integer is returned by INT.

For nonintegers, INT returns the integer closest on the number line to the left of the specified number. Thus, for positive numbers, the decimal portion is dropped; for negative numbers, the next smallest integer is used (i.e., INT(-2.3)=-3).

(number line graphic)

LOG--Natural Logarithm

LOG(numeric-expression)

The natural LOGarithm function returns the natural logarithm of the number specified by the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

The natural logarithm of X is usually shown as log (x). LOG is the inverse
e
of the exponential function (EXP); thus, X=LOG(EXP(X)).

The argument of LOG must be greater than zero. If you specify a value less than or equal to zero, the message BAD ARGUMENT is displayed, and the program stops running.

To find the logarithm of a number in another base B, use the formula,

$$\log_B(X) = \log_e(X)/\log_e(B)$$

For example, $\log_{10}(3) = \log_e(3)/\log_e(10)$

```
>NEW

>100 B=.678
>110 A=INT(B*100+.5)/100
>120 PRINT A;INT(B)
>130 PRINT INT(-2.3);INT(2.2)

>RUN
  .68  0
 -3  2

** DONE **
```

```
>NEW

>100 A=3.5
>110 PRINT LOG(A);LOG(A*2)
>120 PRINT LOG(EXP(2))
>RUN
  1.252762968  1.945910149
  2.

 ** DONE **


>PRINT LOG(-3)

 *BAD ARGUMENT

>PRINT LOG(3)/LOG(10)
  .4771212547
```

RANDOMIZE

RANDOMIZE [seed]

The RANDOMIZE statement is used in conjunction with the random-number function (RND). The seed may be any numeric expression.

If you use the RANDOMIZE statement with a seed specified, the sequence of random numbers generated by RND depends upon the value of the seed. If the same seed is used each time the program is run, the same sequence of numbers is generated; if a different seed is used each time the program is run, a different sequence of numbers is generated.

When the RANDOMIZE statement is used without a seed, a different and unpredictable sequence of random numbers is generated by RND each time the program is run.

When the RANDOMIZE statement is not used, RND generates the same sequence of pseudo-random numbers each time a program is run.

The computer may generate the same sequence of numbers even if you specify different numeric expressions for the seed. The number actually used for the seed is the first two bytes of the internal representation of the number. For example, the first two bytes of the internal representation of 1000 and 1099 are the same and thus they will produce the same seed, which will, in turn, produce the same sequence of numbers. (See "Accuracy Information" in Appendix XXX for more information.)

```
>NEW

>100 RANDOMIZE 23
>110 FOR K=1 TO 5
>120 PRINT INT(10*RND)+1
>130 NEXT K
>RUN
  6
  4
  3
  8
  8

** DONE **
```

RND--Random Number

RND

The random-number function (RND) returns the next pseudo-random number in the current sequence of pseudo-random numbers. The random number generated is greater than or equal to zero and less than 1.

The same sequence of random numbers is generated by RND every time the same program is run unless the RANDOMIZE statement appears in the program.

To obtain random integers from value A through value B (where A<B), use this formula:

```
INT((B-A+1)*RND)+A
```

```
>NEW

>100 FOR K=1 TO 5
>110 RRINT RND
>120 NEXT K
>RUN
  .6XXXXX  CHECK W/ SQUIRREL  XXX
  .4
  .6
  .4
  .3

 ** DONE **


>NEW

>100 FOR K=1 TO 5
>110 C=INT(20*RND)+1
>120 PRINT C
>130 NEXT K
>RUN
  11
  8
  11
  8
  6

 ** DONE **
```

SGN--Signum (Sign)

SGN(numeric-expression)

The signum function (SGN) returns a value representing the algebraic sign of the value specified by the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

SGN gives different values depending on the value of the argument; for the argument X,

X<0,SGN(X)=-1

X=0,SGN(X)=0

X>0,SGN(X)=1

SIN--Sine

SIN(numeric-expression)

The SINe function returns the sine of the argument X, where X is an angle in radians. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

If the angle is given in degrees, multiply the degrees by pi/180 to find the equivalent angle in radians. You may use (4*ATN(1))/180 or 0.0174532925194 for pi/180. Note that if you enter a value of X, where

$|X| \geq 1.570796326637 5 * 10^{10}$, the message BAD ARGUMENT is displayed, and the program stops running.

```
>NEW

>100 A=-23.7
>110 B=6
>120 PRINT SGN(A);SGN(0);SGN(
 B)
>130 PRINT SGN(-3*3);SGN(B*2)

>RUN
 -1  0  1
 -1  1

** DONE **
```

```
>NEW

>100 A=.5235987755982
>110 B=30
>120 C=.01745329251994
>130 PRINT SIN(A);SIN(B*C)
>140 PRINT SIN(B*(4*ATN(1))/1
 80)
>RUN
  .5  .5
  .5

 ** DONE **
```

```
>PRINT SIN(1.9E12)

 * BAD ARGUMENT
```

SQR--Square Root

SQR(numeric-expression)

The square root function (SQR) returns the positive square root of the value specified by the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

SQR(X) is equivalent to X^(1/2).

If the value specified by the argument is negative, the message BAD ARGUMENT is displayed, and the program stops running.

TAN--Tangent

TAN(numeric-expression)

The TANgent function returns the tangent of the argument X, where X is an angle in radians. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for evaluating numeric expressions are used.

If the angle is given in degrees, multiply the degrees by pi/180 to find the equivalent angle in radians. You may use (4*ATN(1))/180 or 0.0174532925199 4 for pi/180. Note that if you enter a value of X where

$$|X| \geq 1.570796326637 5 \times 10^{10}$$

, the message BAD ARGUMENT is displayed, and the program stops running.

BUILI=IN SIRING EUNCIIONS

String functions manipulate strings to produce either a numeric result or a string result. As you use your computer, you will find many ways to use the string functions described. Note that any string function with a name that ends with a dollar sign (for example, CHR$) always returns a string result and therefore cannot be used in numeric expressions.

```
>NEW

>100 PRINT SQR(4);4^(1/2)
>110 PRINT SQR(10)
>RUN
  2  2
  3.16227766

 ** DONE **

>PRINT SQR(-5)

 * BAD ARGUMENT
```

```
>NEW

>100 A=.7853981633973
>110 B=45
>120 C=.0174532925l994
>130 PRINT TAN(A);TAN(B*C)
>140 PRINT TAN(B*(4*ATN(1))/1
 80)
>RUN
  1.  1.
   1

 ** DONE **

>PRINT TAN(1.76E10)

 * BAD ARGUMENT
```

ASC--ASCII Value

ASC(string-expression)

The ASCII value function (ASC) returns the ASCII character code corresponding to the first character in the string-expression. A list of the ASCII character codes for each character in the standard character set is given in the Appendix XX.

CHR$--Character

CHR$(numeric-expression)

The character function (CHR$) returns the character corresponding to the ASCII character code specified in the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

If the argument is not an integer, it is rounded to an integer.

An argument from 32 through 127 gives the standard ASCII character corresponding to that value. Other values give the special graphics symbols. Refer to Appendix XX for a list of the ASCII character codes and their assigned characters.

If a character code is not defined, the character given is whatever is in memory at that location at that time. Any argument that is outside the ASCII character code range is repeatedly reduced by 256 until it is less than 256.

An argument less than zero or greater than 32767 causes the message BAD VALUE to be displayed, and the program stops running.

```
>NEW

>100 A$="HELLO"
>110 C$="JACK SPRAT"
>120 B$="THE ASCII VALUE OF "
>130 PRINT B$;"H IS";ASC(A$)
>140 PRINT B$;"J IS";ASC(C$)
>150 PRINT B$;"N IS";ASC("NAM
 E")
>160 PRINT B$;"1 IS";ASC("1")
>170 PRINT CHR$(ASC(A$))
>RUN
 THE ASCII VALUE OF H IS 72
 THE ASCII VALUE OF J IS 74
 THE ASCII VALUE OF N IS 78
 THE ASCII VALUE OF 1 IS 49
 H

 ** DONE **


>NEW

>100 A$=CHR$(72)&CHR$(73)&CHR
 $(33)
>110 PRINT A$
>120 PRINT CHR$(3*14)
>130 PRINT CHR$(ASC("┼"))
>RUN
 HI!
 *
 ┼

 ** DONE **


>PRINT CHR$(33010)

 * BAD VALUE
```

LEN--Length

LEN(string-expression)

The LENgth function returns the number of characters in the string specified by the argument. The argument is the string value obtained when the string-expression is evaluated. The normal rules for the evaluation of string expressions are used.

The length of a null string is zero. Bear in mind that a space is a character and counts as part of the length.

POS--Position

POS(string1,string2,numeric-expression)

The POSition function finds the first occurrence of string2 within string1. Both string1 and string2 are string expressions. The numeric-expression is evaluated and rounded, if necessary, to the nearest integer, N. The normal rules for the evaluation of string expressions and numeric expressions are used.

The search for string2 begins at the Nth character of string1.

If string2 is found, the character position within string1 of the first character of string2 is given.

If string2 is not found, a value of zero is given.

The position of the first character in string1 is position one. If N is greater than the number of characters in string1, a value of zero is given. If N is less than zero, the message BAD VALUE is displayed, and the program stops running.

```
>NEW

>100 NAME$="CATHY"
>110 CITY$="NEW YORK"
>120 MSG$="HELLO "&"THERE!"
>130 PRINT NAME$;LEN(NAME$)
>140 PRINT CITY$;LEN(CITY$)
>150 PRINT MSG$;LEN(MSG$)
>160 PRINT LEN(NAME$&CITY$)
>170 PRINT LEN("HI!")
>RUN
 CATHY  5
 NEW YORK  8
 HELLO THERE! 12
  13
  3

 ** DONE **


>NEW

>100 MSG$="HELLO THERE! HOW A
 RE YOU?"
>110 PRINT "H";POS(MSG$,"H",1
 )
>120 C$="RE"
>130 PRINT C$;POS(MSG$,C$,1);
 POS(MSG$,C$,12)
>140 PRINT "HI";POS(MSG$,"HI"
 ,1)
>RUN
 H 1
 RE  10  19
 HI  0

 ** DONE **
```

SEG$--String Segment

SEG$(string-expression,numeric-expression1,numeric-expression2)

The string SEGment function returns a portion (substring) of the string designated by the string-expression. The normal rules for the evaluation of numeric expressions and string expressions are used.

Numeric-expression1 specifies the position of the character in the string-expression that is to be the first character of the substring. The position of the first character in the string-expression is position one. Numeric-expression2 specifies the length of the substring.

In this example, A$ is string-expression, X is numeric-expression1, and Y is numeric-expression2. If you specify either a value for X greater than the length of A$ (line 110) or a value of zero for Y (line 120), the program returns a null string. If you specify a value for Y greater than the remaining length in A$ starting at the position specified by X (line 130), the substring is the remainder of A$ from position X on.

If X is less than or equal to zero or if Y is less than zero, the message BAD VALUE is displayed, and the program stops running.

```
>NEW

>100 MSG$="HELLO THERE! HOW A
 RE YOU?"
>110 REM SUBSTRING BEGINS IN
 POSITION 14 AND HAS A LENGTH
 OF 12.
>120 PRINT SEG$(MSG$,14,12)
>RUN
 HOW ARE YOU?

 ** DONE **


>NEW

>100 MSG$="I AM A COMPUTER."
>110 PRINT SEG$(MSG$,20,1)
>120 PRINT SEG$(MSG$,10,0)
>130 PRINT SEG$(MSG$,8,20)
>RUN

 COMPUTER.

 ** DONE **


>PRINT SEG$(MSG$,-1,10)

 * BAD VALUE
```

STR$--String-Number

STR$(numeric-expression)

The STRing-number function returns the string representation of the number specified by the argument. The argument is the value obtained when the numeric-expression is evaluated. The normal rules for the evaluation of numeric expressions are used.

When the number is converted into a string, the string is a valid representation of a numeric constant with no leading or trailing spaces. For example, if B=69.5, the STR$(B) is the string "69.5". Only string operations may be performed on the strings created using STR$.

The STR$ is the inverse of the value function (VAL).

VAL--Value

VAL(string-expression)

The VALue function returns the numeric constant that results when the string-expression is converted to a number. For example, VAL converts the string "69.5" to the numeric constant 69.5. The normal rules for the evaluation of string expressions are used.

If the string-expression is not a valid representation of a number or if the string-expression is of zero length, the message BAD ARGUMENT is displayed and the program stops running. If the string-expression is longer than 255 characters, the message BAD ARGUMENT is displayed and the program execution stops.

VAL is the inverse of the string-number function (STR$).

```
>NEW

>100 A=-26.3
>110 PRINT STR$(A);" ";A
>120 PRINT 15.7;STR$(15.7)
>130 PRINT STR$(VAL("34.8"))
>RUN
 -26.3 -26.3
  15.7 15.7
 34.8

 ** DONE **
```

```
>NEW

>100 P$="23.6"
>110 N$="-4.7"
>120 PRINT VAL(P$);VAL(N$)
>130 PRINT VAL("52"&".5")
>140 PRINT VAL(N$&"E"&"12")
>150 PRINT STR$(VAL(P$))
>RUN
  23.6 -4.7
  52.5
 -4.7E+12
 23.6

 ** DONE **
```

USER DEFINED FUNCTIONS

In addition to the built-in functions described in the two previous sections, TI-99/2 BASIC enables you to define your own functions to use within a program. User-defined functions can simplify programming by avoiding repeated use of complicated expressions. Once a function has been defined using the DEF statement, it may be used anywhere in the program by referencing the name you gave to the function.

DEF function-name[(parameter)] = expression

The DEFine statement enables you to define your own functions to use within a program. The function-name may be any valid variable name; it is assigned the value of the expression. If the expression evaluates to a string, the function-name must be a string-variable name (one that ends with $).

The parameter is used to pass information to the DEF statement. If a parameter is specified, it must be a valid variable name enclosed in parentheses following the function-name.

The function-name, like any variable or built-in function, can be used in expressions. However, the function-name is assigned a value only when an expression that contains function-name is evaluated.

When a function-name is defined in the DEF statement with no parameter, the function-name is assigned the value of expression, using the current values of the variables that appear in the expression.

When a function-name is defined in the DEF statement with a parameter, an argument enclosed in parentheses must follow the function-name. The parameter is assigned the value of the argument. The expression is then evaluated using the newly assigned value of the parameter and the current values of any other variables in the DEF statement.

The variable name used for a parameter is local to the DEF statement in which it is used. Therefore, if a variable in the program has the same name as a parameter, the value of the variable is not affected when the parameter is assigned the value of the argument.

When the computer encounters a DEF statement, it takes no action but proceeds to the next statement.

A DEF statement may appear anywhere in a program, but it must be executed to define a function before you can call that function.

A DEF statement can reference other defined functions (line 170).

In a DEF statement, the function-name may not reference itself either directly (e.g. DEF B=B*2) or indirectly (e.g. DEF F=G; DEF G=F).

The parameter cannot be an array name. You can use an array element in a DEF statement if that element does not have the same name as the parameter.

```
>NEW

>100 DEF PI=4*ATN(1)
>110 PRINT COS(60*PI/180)
>RUN
  .5

 ** DONE **


>NEW

>100 REM EVALUATE Y=X*(X-3)
>110 DEF Y=X*(X-3)
>120 PRINT " X  Y"
>130 FOR X=-2 TO 5
>140 PRINT X;Y
>150 NEXT X
>RUN
  X  Y
 -2  10
 -1  4
  0  0
  1 -2
  2 -2
  3  0
  4  4
  5  10

 ** DONE **


>NEW

>100 REM TAKE A NAME AND PRIN
 T IT BACKWARDS
>110 DEF BACK$(X)=SEG$(NAME$,
 X,1)
>120 INPUT "NAME? ":NAME$
>130 FOR J=LEN(NAME$) TO 1 ST
 EP -1
>140 BNAME$=BNAME$&BACK$(J)
>150 NEXT J
>160 PRINT NAME$:BNAME$
>RUN
 NAME?  ROBOT
 ROBOT
 TOBOR

 ** DONE **
```

```
>NEW

>100 DEF FUNC(A)=A*(A+B-5)
>110 A=6.9
>120 B=13
>130 PRINT "B=";B;"FUNC(3)=";
 FUNC(3);"A=";A
>RUN
 B= 13
 FUNC(3)= 33
 A= 6.9

 ** DONE **
```

```
>NEW

>100 REM FIND F'(X) USING NUM
 ERICAL APPROXIMATION
>110 INPUT "X=? ":X
>120 IF ABS(X)>.01 THEN 150
>130 H=.00001
>140 GOTO 180
>150 H=.001*ABS(X)
>160 DEF F(Z)=3*Z^3-2*Z+1
>170 DEF DER(X)=(F(X+H)-F(X-H
 ))/(2*H)
>180 PRINT "F'(";STR$(X);")="
 ;DER(X)
>RUN
 X=? .1
 F'(.1)= -1.90999997

 ** DONE **
```

```
>NEW

>100 DEF GX(X)=GX(2)*X
>110 PRINT GX(3)
>RUN

 * MEMORY FULL IN 110

>100 DEF GX(A)=A(3)^2
>RUN

 * NAME CONFLICT IN 100
```

```
>NEW

>100 DEF SQUARE(X)=X*X
>110 PRINT SQUARE
>RUN

 * NAME CONFLICT IN 110

>100 DEF PI=3.1416
>110 PRINT PI(2)
>RUN
```

ARRAYS

Arrays are collections of variables that are arranged for easy use in a computer program. The most common use of an array is to store values that are in a list; a one-dimensional array is used for a list. A two-dimensional array can be used to store the values of a table.

You can use arrays with one, two, or three dimensions in TI-99/2 BASIC.

Each variable in the array is called an element. The size of an array is limited only by the amount of memory available.

By using the array capabilities of TI-99/2 BASIC, you can do many useful things such as printing the elements forward or backward, rearranging them, adding them together, multiplying them, or processing selected elements.

OPTION BASE [0 or 1]

The OPTION BASE statement enables you to set the lower limit of array subscripts at 1 instead of 0. You can omit the OPTION BASE statement if you want the lower limit of the subscripts to be 0.

If you include an OPTION BASE statement in your program, you must give it a lower line number than any DIMension statement or any reference to an element in any array. You may have only one OPTION BASE statement in a program, and it applies to all array subscripts in your program. Therefore, you cannot have one array subscript beginning with 0 and another beginning with 1 in the same program.

If you use some integer other than 1 or 0 in the OPTION BASE statement, the computer stops the program and prints INCORRECT STATEMENT.

```
>NEW

>100 OPTION BASE 1
>110 DIM X(5,5,5)
>120 X(1,0,1)=3
>130 PRINT X(1,0,1)
>RUN

 * BAD SUBSCRIPT IN 120

>100 (Press ENTER to delete line 100.)
>RUN
  3

** DONE **
```

DIMension

DIM array-name (integer1[,integer2][,integer3])[,array-name . . .]

The DIMension statement reserves space for both numeric and string arrays. Once used, an array-name cannot appear in another DIM statement in the same program.

A DIM statement is required for any array used in a program and must appear in the program before any other reference to the array-name. An array-name must be a valid variable name. Multiple array-names in a DIM statement must be separated by commas.

You may use one-, two-, or three-dimensional arrays in TI-99/2 BASIC. The number of integers in parentheses following the array-name tells the computer how many dimensions the array has.

A one-dimensional array-name is followed by one integer, which specifies the number of values in the array.

A two-dimensional array-name is followed by two integers, which define the number of rows and columns in the array.

A three-dimensional array-name is followed by three integers, which define the number of rows, columns, and pages in the array.

Thus,

```
!o! DIM A(6)--describes a one-dimensional array
!o! DIM A(12,3)--describes a two-dimensional array
!o! DIM A(5,2,11)--describes a three-dimensional array
```

An array is allocated space after you enter the RUN command but before the program is actually run. However, until you place values in an array, each element in a string array is a null string and each element in a numeric array has a value of zero.

If your computer cannot reserve space for an array with the dimensions you specify, a MEMORY FULL message is displayed, and your program will not run.

```
>DIM A(12),B(5)
```

```
>NEW

>100 DIM X(15)
>110 FOR K=1 TO 15
>120 READ X(K)
>130 NEXT K
>140 REM PRINT LOOP
>150 FOR K=15 TO 1 STEP -1
>160 PRINT X(K);
>170 NEXT K
>180 DATA 1,2,3,4,5,6,7,8,9,1
 0,11,12,13,14,15
>RUN
  15  14  13  12  11  10  9
  8  7  6  5  4  3  2  1
 ** DONE **
```

DIM

Subscripting An Array

To reference an array in a program, you must specify which element in the array the computer is to use. You can specify the element by using a subscript. Subscripts are enclosed in parentheses immediately following the name of the array. A subscript must be a valid numeric expression that evaluates to a non-negative result. This result is rounded to the nearest integer, if necessary.

The number of elements reserved for an array determines the maximum value of each subscript for that array.

The example on the right assumes that the array begins with element 1 (OPTION BASE 1 on line 120):

- line 130--This line defines T as a one-dimensional array with 25 elements.
- line 160--The numeric variable I subscripts T. Whatever value I contains at this time is used to point to an element of T. If I=3, the third element of T is added.
- line 200--The subscript 14 tells the computer to print the fourteenth element of T.
- line 220--The computer evaluates the numeric expression N+2. If N=15 at this time, the seventeenth element of T is printed.

If you access an array with a subscript greater than the maximum number of elements defined for that array, or if a subscript has a 0 value and you have used an OPTION BASE 1 statement, a BAD SUBSCRIPT message is displayed, and the program ends.

```
>NEW

>100 REM DEMO OF DIM AND SUBS
 CRIPTS
>110 S=100
>120 OPTION BASE 1
>130 DIM T(25)
>140 FOR K=1 TO 25
>150 READ T(K)
>160 A=S+T(K)
>170 PRINT A;
>180 NEXT K
>190 PRINT::
>200 PRINT T(14)
>210 INPUT "ENTER A NUMBER BE
 TWEEN 1 AND 23:":N
>220 PRINT T(N+2)
>230 DATA 12,13,43,45,65,76,7
 8,98,56,34,23,21,100,333,222
 ,111,444,666,543,234,89,765,
 90,101,345
>RUN
  112  113  143  145  165
  176  178  198  156  134
  123  121  200  433  322
  211  544  766  643  334
  189  865  190  201  445

  333
ENTER A NUMBER BETWEEN 1 AND
 23:14
  111

** DONE **
```

SUBROUTINE_STATEMENTS

Subroutines may be thought of as separate, self-contained programs within a main program. Subroutines perform tasks, such as printing information, performing calculations, or reading values into an array. Using a subroutine enables you to type a set of statements only once and then access it (with a GOSUB statement) at any point in the program.

GOSUB

GOSUB

line-number

GO SUB

The GOSUB statement is used with the RETURN statement to branch to a subroutine, perform the steps in the subroutine, and return to the next program line following the GOSUB statement. When the computer performs the GOSUB statement, it stores the next line number of the main program; the computer returns to that point when it encounters a RETURN statement in the subroutine.

```
>NEW

>100 FOR X=1 TO 10
>110 GOSUB 150
>120 PRINT "X =";X;"SO X IS ";A$
>130 NEXT X
>140 STOP
>150 IF X/2=INT(X/2) THEN 180
>160 A$="ODD"
>170 RETURN
>180 A$="EVEN"
>190 RETURN
>RUN
 X =1 SO X IS ODD
 X =2 SO X IS EVEN
 X =3 SO X IS ODD
 X =4 SO X IS EVEN
 X =5 SO X IS ODD
 X =6 SO X IS EVEN
 X =7 SO X IS ODD
 X =8 SO X IS EVEN
 X =9 SO X IS ODD
 X =10 SO X IS EVEN

** DONE **
```

Within a subroutine, the computer can jump to another subroutine, perform it, return to the first subroutine, finish its steps, and then return to the main program at the point where the original branch occurred. GOSUB and RETURN statements must be properly paired; be sure to exercise care in designing subroutines so that the computer will not lose its place.

In the example on the right, the main program jumps to subroutine 1 when it reaches line 500. In subroutine 1, when the program reaches line 730, it jumps to subroutine 2. When the RETURN in subroutine 2 is encountered (line 850), the computer returns to subroutine 1 at line 740, finishes the subroutine, returns to the main program, and completes it through line 600.

If the GOSUB statement transfers program control to a line-number not in the program, the program ends and the message BAD LINE NUMBER is displayed. If the GOSUB transfers the program control to its own line-number, the program stops and the message MEMORY FULL is displayed

```
>NEW

>100 REM NESTED SUBROUTINES
>110 REM MAIN PROGRAM
      .
      . (Program lines . . .)
      .
>500 GOSUB 700
>510 .
      .
      .
>600 STOP
>700 REM SUBROUTINE1
      .
      .
      .
>730 GOSUB 800
>740 .
      .
      .
>790 RETURN
>800 REM SUBROUTINE2
      .
      .
      .
>850 RETURN
```

```
>NEW

>100 X=12
>110 Y=23
>120 GOSUB 120
>130 PRINT Z
>140 STOP
>150 REM SUBROUTINE
>160 Z=X+Y*120/5
>170 RETURN
>RUN

* MEMORY FULL IN 120


>120 GOSUB 150
>RUN
  564

** DONE **
```

RETURN

The RETURN statement is used with the GOSUB statement to provide a branch-and-return structure. When the computer encounters a RETURN statement, it takes program control back to the program line immediately following the GOSUB statement that transferred the computer to that particular subroutine.

You can develop programs with subroutines that jump to other subroutines and back again, if you make sure that each GOSUB leads the computer to a RETURN statement.

If the computer encounters a RETURN statement before performing a GOSUB instruction, the program stops, and the message CAN'T DO THAT appears.

STOP

STOP

The STOP statement terminates the program that is running. STOP can be used interchangeably with END. You can place one or more STOP statements anywhere in your program. Normally, the STOP statement is used when there are several ending points in a program and the END statement is used when there is only one ending point.

```
>NEW

>100 FOR K=1 TO 3
>110 GOSUB 150
>120 PRINT "K=";K
>130 NEXT K
>140 STOP
>150 REM SUBROUTINE
>160 FOR X=1 TO 2
>170 PRINT "X=";X
>180 NEXT X
>190 RETURN
>RUN
 X= 1
 X= 2
 K= 1
 X= 1
 X= 2
 K= 2
 X= 1
 X= 2
 K= 3

 ** DONE **
```

```
>NEW

>100 A=5
>110 B$="TEXAS INSTRUMENTS"
>120 PRINT B$;A
>130 STOP
>RUN
 TEXAS INSTRUMENTS 5

 ** DONE **
```

ON GOSUB

```
                          GOSUB
ON numeric-expression          line-number [,line-number] . . .
                         GO SUB
```

The ON GOSUB statement instructs the computer to perform a subroutine, depending on the value of the numeric-expression. The computer first evaluates the numeric-expression and converts the result to an integer, rounding if necessary.

!o! If the integer is 1, the computer branches to the first line-number listed in the ON-GOSUB statement.

!o! If the integer is 2, the computer branches to the second line-number listed, and so on.

The computer saves the number of the line following the ON GOSUB statement and returns to this point after performing the subroutine. The subroutine must contain a RETURN statement to signal the computer to go back to the saved line number and continue the program from that statement.

If the subroutine does not contain a RETURN statement, the program continues as if a GOTO had been performed instead of a GOSUB.

If the rounded value of the numeric-expression is less than 1 or greater than the number of line-numbers in the ON GOSUB statement, the program stops, and the message BAD VALUE IN line-number appears. In the example, the original line 130 transfers control to the END statement if a 9 is input for CODE, allowing the program to end without an error message.

If a line-number in the ON GOSUB statement is not a valid program line, the message BAD LINE NUMBER is displayed.

```
>NEW

>100 INPUT "CODE=?":CODE
>110 IF CODE=9 THEN 290
>120 INPUT "HOURS=?":HOURS
>130 ON CODE GOSUB 170,200,23
 0,260
>140 PAY=RATE*HOURS+BASEPAY
>150 PRINT "PAY IS $";PAY
>160 GOTO 100
>170 RATE=3.10
>180 BASEPAY=5
>190 RETURN
>200 RATE=4.25
>210 BASEPAY=25
>220 RETURN
>230 RATE=10
>240 BASEPAY=50
>250 RETURN
>260 RATE=25
>270 BASEPAY=100
>280 RETURN
>290 END
>RUN
 CODE=?4
 HOURS=?40
 PAY IS $ 1100
 CODE=?2
 HOURS=?37
 PAY IS $ 182.25
 CODE=?3
 HOURS=?35.75
 PAY IS $ 407.5
 CODE=?1
 HOURS=?40
 PAY IS $ 129
 CODE=?9

 ** DONE **

>RUN
 CODE=?5
 HOURS=?40

 * BAD VALUE IN 130

>130 ON CODE GOSUB 170,200,23
 0,600
>RUN
 CODE=?4
 HOURS=?40

 * BAD LINE NUMBER IN 130
```

FILE PROCESSING STATEMENTS

Your computer has the ability to store both programs and data on peripheral (accessory) devices. You can later load and use these files with your computer as often as you wish and delete them when you no longer need them.

With the file-processing capability of your computer, you can save important information, create procedures to update data, and avoid retyping your programs. TI-99/2 BASIC provides an extensive range of file-processing features, including sequential and relative file organization and processing, fixed and variable length records, and display and internal formats for data.

The connecting device between the Basic Computer 99/2 and the family of HEX-BUS™ peripherals is the HEX-BUS interface that is built into the computer itself. The HEX-BUS peripherals include the Texas Instruments Wafertape™ Digital Tape Drive, the Printer/Plotter, and the RS232 Interface/Parallel Port. The computer identifies these peripherals by the device numbers listed below.

Device	Device Number
Wafertape™ Drive	1--8
Printer/Plotter	10--17
RS232 Interface	20--27
Parallel Output Port	50--51

Other HEX-BUS peripherals will be available in the future. Check with your dealer for a complete list of HEX-BUS peripherals.

OPEN

OPEN #file-number: "HEXBUS.device-number.filename" [,open-mode]

"CS1"

[,file-type] [,file-organization] [,record-type] [,file-life]

The OPEN statement enables a BASIC program to process a file. OPEN assigns a file-number to a file on a peripheral, thereby establishing the link between that file and its file-number; this number is used by all Input/Output statements that refer to the file.

- File-number--The file-number is a numeric expression that, when evaluated and rounded, must be a number from 1 through 255. The file-number must be preceded by a number sign (#). All programs that access files must use the OPEN statement to assign a file-number to a file or device.

  File-number 0 refers to the keyboard and screen of your computer and is always accessible. You cannot open or close file-number 0.

  Each file in your program must have its own separate number. If a file-number specifies a file that is already open, an error occurs.

Using the OPEN Statement with HEX-BUS Peripherals

OPEN associates a file-number with the specified filename on a peripheral; therefore, when an Input/Output statement uses a file-number, the computer knows which file to access. Before a statement can use a file-number to access a file on a HEX-BUS peripheral, an OPEN statement must have first assigned that file-number to the filename.

The OPEN statement describes a file's characteristics so that your program can create or process the file. The keyword HEXBUS must be included in any OPEN statement that refers to HEX-BUS™ peripherals. When you open an existing file on a HEX-BUS peripheral, the computer checks to see if the file or device characteristics match the information specified in the OPEN statement for that file. If they do not match, the file is not opened, and an I/O error message is displayed. If the computer cannot find a file that is to be opened in INPUT mode, an I/O error message is displayed.

- Device-number--The device-number is a number from 1 through 255 by which the computer identifies a peripheral. For example, 20 is the device-number for the RS232 Interface peripheral.

- Filename--A filename supplies information to the peripheral device for the OPEN statement. For example, with an external storage device, filename specifies the name of the file. With other devices, the filename specifies options such as parity, baud rate, etc. If you use a string constant as a filename, you must enclose it in quotation marks.

Refer to the individual HEX-BUS peripheral manuals for more information about the device-number and for specific information about the form of a filename.

OPEN

```
XX 100 OPEN #2:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED 128,PE
 RMANENT
```

XX Need examples for opening HEX-BUS peripherals.XX

```
>100 OPEN #25:"CS1",SEQUENTIA
 L,INTERNAL,INPUT,FIXED,PERMA
 NENT
>110 X=100
>120 N=2
>130 OPEN #122:"CS"&STR$(N),S
 EQUENTIAL,INTERNAL,OUTPUT,FI
 XED,PERMANENT
```

OPEN

The characteristics listed below may be in any order or may be omitted. When a characteristic is omitted, the computer assumes standard characteristics called defaults.

!o! open-mode--This entry specifies in which of the following modes the computer is to process the file. If the open-mode is omitted, the computer opens the file in UPDATE mode.

INPUT--The computer can only read from the file.

OUTPUT--The computer can only write to the file.

UPDATE--The computer can both read from and write to the file.

APPEND--The computer can write data only at the end of the file; the records already on the file cannot be accessed.

!o! file-type--This specification designates the format of the data stored on the file. Data stored in ASCII characters (the kind displayed on the screen) are called DISPLAY. Each DISPLAY record usually corresponds to one print line.

Data stored in the internal machine format (binary code) are called INTERNAL. The INTERNAL format is more efficient for recording data on mass storage devices; it requires less space and less processing time because the computer performs fewer conversions between formats.

If the file-type is omitted, the computer assumes DISPLAY format.

!o! file-organization--Files can be organized either sequentially or randomly. Records on a SEQUENTIAL file are read or written one after the other in sequence from beginning to end. Note that files on the Wafertape™ peripheral must have SEQUENTIAL organization. Random-access files (called RELATIVE in TI-99/2 BASIC) can be read or written in any record order; they can also be processed sequentially.

To indicate the organization of a file, specify either SEQUENTIAL or RELATIVE in the OPEN statement. If file-organization is omitted, the computer assumes SEQUENTIAL organization.

You may optionally include the initial number of records on a file by following the word SEQUENTIAL or RELATIVE with a numeric expression.

!o! record-type--This entry specifies whether the records on the file are all the same length (FIXED) or vary in length (VARIABLE). The keyword FIXED or VARIABLE may be followed by a numeric expression specifying the maximum length of a record. The maximum length of a record varies with the device used.

TM

In the example on the right, the file MYFILE on device 1 (a Wafertape drive) is opened as file number 4. Data can only be written to the file. If the file does not already exist, the file is created with the characteristics created in the OPEN statement. If the file already exists, the characteristics of the file are compared with those given in the OPEN statement.

XXXXThis example illustrates opening the file NAME$ on device 100 (that is assumed to support relative files) with the file number 10. The file can only be read.

FILE2 in this example is opened on device 1 as file number 12. Data can only be written at the end of the file. The computer assumes SEQUENTIAL organization and DISPLAY data format.

The file NAME$ on device 1 in UPDATE mode with file number 53. If the file does not already exist, the file is created with the characteristics created in the OPEN statement. If the file already exists, the characteristics of the file are compared with those given in the OPEN statement.

The file NAME$ is opened on device 1 with the file number 11. The file can be read only.

```
>100 OPEN #4:"HEXBUS.1.MYFILE
 ",OUTPUT,INTERNAL
```

XXXX HEX-BUS peripherals do not support relative files XXXX

```
>120 OPEN #10:"HEXBUS.100.NA
 ME$",RELATIVE,INPUT,INTERNAL
```

```
>100 OPEN #12:"HEXBUS.1.FILE2
 ",APPEND,FIXED
```

```
>100 OPEN #53:"HEXBUS.1.NAME
 $",FIXED,INTERNAL
```

```
>100 OPEN #11:"HEXBUS.1.NAME
 S$",INPUT,INTERNAL,SEQUENTI
 AL,VARIABLE 100
```

Using the OPEN Statement with the II Prog m Recorder or a Compatible Recorder

OPEN associates a file-number with the Program Recorder; it does not link file-number with a specific file. You must locate that file on the device. Refer to Book 1 for instructions on locating programs or data on an audio cassette tape. Before a statement can use a file-number to access a cassette device, an OPEN statement must have first associated that file-number with the cassette device.

"CS1" must be included in the OPEN statement. The following characteristics may be in any order, but certain of them are required.

!o! open-mode (required)--This entry specifies in which of the following modes the computer is to process the file.

INPUT The computer can only read from the file.

OUTPUT The computer can only write to the file.

!o! file-type (optional)--This specification designates the format of the data or how the data are recorded on the file.

When data are stored in ASCII characters (the kind displayed on the screen), the data format is called DISPLAY. A DISPLAY record usually corresponds to one print line.

When data are stored in the internal machine format (binary code), the data format is called INTERNAL. INTERNAL format is more efficient for recording data on mass-storage devices; it requires less space and less processing time because the computer performs fewer conversions between formats.

If the file-type is omitted, the computer assumes DISPLAY format.

!o! file-organization (optional)--Files on a cassette recorder must have SEQUENTIAL organization; this specification may be ommitted, because the computer assumes SEQUENTIAL organization.

!o! record-type (required)--You must specify FIXED. This entry indicates that all the records on the file are the same length (FIXED). The keyword FIXED may be followed by a numeric expression specifying the maximum length of a record. You may specify any length up to 192 positions. If the length specification is omitted, the computer assumes a length of 64 positions.

!o! file-life (optional)--Files you create with your TI Computer are considered PERMANENT, not temporary; if this entry is omitted, the computer assumes a PERMANENT file-life.

```
>OPEN #75:"CS1",OUTPUT,FIXED
```

The file located at the current position of the cassette tape is opened as file number 75. Data can only be written to the file. The computer assumes a SEQUENTIAL file in DISPLAY format with a FIXED length of 64 characters.

The file located at the current position of the cassette tape is opened as file number 2. When the computer performs the OPEN statement, the instructions for activating a cassette recorder device are displayed.

```
>NEW

>100 OPEN #2:"CS1",INTERNAL,I
 NPUT,FIXED
  .
  . (Program lines . . .)
  .
>300 CLOSE #2
>RUN

  * REWIND CASSETTE TAPE    CS1
    THEN PRESS ENTER

  * PRESS CASSETTE PLAY     CS1
    THEN PRESS ENTER

  .
  . (Rest of program run.)
  .
  * PRESS CASSETTE STOP     CS1
    THEN PRESS ENTER

  ** DONE **
```

INPUT

INPUT #file-number[,REC numeric-expression][:variable-list]

(See also the "Input-Output Statements" section)

This form of the INPUT statement enables you to read data from a peripheral device. The INPUT statement can be used only with files opened in INPUT or UPDATE mode.

The file-number must be the file-number of a currently open file. (See the OPEN statement.) File-number 0 is the keyboard and may always be used. If you use file-number 0, the INPUT statement is performed as described in "Input-Output Statements," except that you cannot specify an input-prompt.

The variable-list contains variables that are assigned values when the INPUT statement is performed. Variable names in the variable-list are separated by commas and may be numeric and/or string variables.

Filling the variable-list

When the computer reads records from a file, it stores each complete record internally in a temporary storage area called an input/output (I/O) buffer. A separate buffer is provided for each open file-number. Values are assigned to variables in the variable-list from left to right, using the data in this buffer. When a variable-list has been filled with corresponding values, any data items left in the buffer are discarded unless the INPUT statement ends with a trailing comma, which creates a "pending" input condition (see "Using Pending Inputs").

If the variable-list is longer than the number of data items in the current record being processed, the computer gets the next record from the file and uses its data items to complete the variable-list, as shown on the right.

```
>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED
>110 INPUT #13:A,B,C$,D$,X,Y,
 Z$
>120 IF A=99 THEN 150
>130 PRINT A;B:C$:D$,X,Y:Z$
>140 GOTO 110
>150 CLOSE #13
>RUN

(The data stored on tape are printed on the screen.)

 ** DONE **


>NEW

>100 OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,C,D
.
. (Program lines . . .)
.
>300 CLOSE #13
>RUN

(First INPUT RECORD=22,77,56,92.
Results:)
A=22  B=77  C=56  D=92

 ** DONE **

>NEW

>100  OPEN #13:"CS1",SEQUENTIA
 L,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,C D,E,F,G
.
.
>CLOSE #13
>RUN
(1st INPUT RECORD=22,33.5
 2nd INPUT RECORD=405,92
 3rd INPUT RECORD=22,11023
 4th INPUT RECORD=99,100

 RESULTS:)

 A=22 B=33.5 C=405 D=92
 E=22 F=11023 G=99

 ** DONE **
```

INPUT

The computer interprets DISPLAY and INTERNAL data differently.

DISPLAY data have the same form as data entered f m the keyboard. The computer knows the length of each data item in a DISPLAY record by the comma separators placed between items. Leading and trailing spaces are ignored unless they are enclosed in quotation marks in a string value. When the computer encounters two adjacent commas, a null string is assigned to the variable.

Each value is checked to ensure that numeric values are placed in numeric variables, as shown on the right in Record 1. If the value (as in Record 2 on the right, JG) is not a numeric value, an INPUT ERROR occurs and the program stops.

INTERNAL data have the following form:

<u>Numeric Items</u>: (insert graphic on page II-126)
designates length of item (always 8) value of item

<u>String Items</u>: (insert graphic on page II-126)
designates length of item value of item

The computer determines the length of each INTERNAL data item by interpreting the one-position length indicator at the beginning of each item.

Limited validation of INTERNAL data items is performed. All numeric items must be 9 positions long (8 digits plus one position that specifies the length) and must be valid representations of floating-point numbers. Otherwise, an INPUT ERROR occurs, and the program stops.

For FIXED-length INTERNAL records, reading beyond the actual data recorded in each record causes padding characters (binary zeros) to be read. If you attempt to assign these characters to a numeric variable, an INPUT ERROR occurs. If strings are being read, a null string is assigned to the string variable.

```
>NEW

>100 OPEN #13:"CS1"SEQUENTIAL
 ,DISPLAY,INPUT,FIXED 64
>110 INPUT #13:A,B,STATE$,D$,
 X,Y

 (INPUT RECORD 1 = 22,97.6,TEXAS,"AUTO LICENSE",22000,-.07
  INPUT RECORD 2 = JG,22,TEXAS,PROPERTY TAX,42,15)
```

INPUT

Using_INPUT_with_RELATIVE_Files

(See the OPEN statement for a description of RELATIVE file-organization.)

You can read RELATIVE files either sequentially or randomly. The computer sets up an internal counter that indicates the record that is to be processed next. The first record in a file is record 0. Thus, the counter begins at zero and is incremented by 1 after every access to the file (whether that access reads or writes a record). In the example on the right, the statements direct the computer to read the file sequentially.

The internal counter can be changed by using the REC clause. The NUMERIC-EXPRESSION following the keyword REC is evaluated to designate a specific record number on the file. When the computer performs an INPUT statement with a REC clause, it reads the specified record from the designated file and places it in the I/O buffer. The REC clause can appear only in statements referencing RELATIVE files. The example on the right illustrates accessing a RELATIVE file randomly, using the REC clause.

If you read and write records on the same file within a program, be sure to use the REC clause. The same internal counter is incremented when records are either read from or written to the same file; you may skip some records and write over others if REC is not used, as shown in the example on the right.

If the internal counter indicates a record beyond the limits of the file that the computer tries to access, the program stops, and an INPUT ERROR message appears.

```
NEW

>100 OPEN #4:"HEXBUS.1.NAME
 $",INTERNAL,INPUT,FIXED 64
>110 INPUT #4:A,B,C$,D$,X
.
. (Program lines . . .)
.
>200 CLOSE #4


>NEW

>100 OPEN #6:"HEXBUS.1.NAME$
 ",INTERNAL,UPDATE,FIXED 72
>110 INPUT K
>120 INPUT #6,REC K:A,B,C$,D$
.
. (Program lines . . .)
.
>300 CLOSE #6
```

```
>NEW

>100 OPEN #3:"HEXBUS.1.NAME$
 ",INTERNAL,UPDATE,FIXED
>110 FOR K=1 TO 10
>120 INPUT #3:A$,B$,C$,X,Y
.
. (Program lines . . .)
.
>230 PRINT #3:A$,B$,C$,X,Y
>240 PRINT NEXT K
>250 CLOSE #3

 (LINE 120--Reads records 0,2,4,6,8...
  LINE 230--Writes records 1,3,5,7,9...)
```

INPUT

Using Pending Inputs

A pending input condition occurs when an INPUT statement ends with a trailing comma. When the computer encounters the next INPUT statement using that file, one of the following actions occurs:

> If the next INPUT statement has no REC clause, the computer uses the data in the I/O buffer beginning where the previous INPUT statement stopped.
>
> If the next INPUT statement includes a REC clause, the computer terminates the pending input condition and reads the specified record into the file's I/O buffer.

If you use a pending input with *file-number* 0, the error message INCORRECT STATEMENT is displayed, and the program stops running.

EOF--End-of-file

EOF(*file-number*)

In sequential processing, the End-Of-File function (EOF) can be used to determine whether you have reached the end of a file. If you attempt to read past the end of a file, an error occurs.

The *file-number* is a numeric-expression that evaluates to the number used in the OPEN statement to open that file. The normal rules for the evaluation of numeric expressions are used.

The EOF function can be used in an IF THEN statement before an INPUT statement reads that file. The EOF function is used to determine if any data remain in the file. The value returned by EOF depends on the position of the file.

Value	Position
0	Not end-of-file
+1	Logical end-of-file
-1	Physical end-of-file

A file is positioned at the logical end-of-file when all records on the file have been accessed. A file is positioned at a physical end-of-file when no more space is available for the file.

```
>NEW

>100 INPUT #0:A,B,
>110 PRINT A;B
>120 GOTO 100
>RUN
  ?
  * INCORRECT STATEMENT
      IN 100



>NEW

>100 OPEN #5:"HEXBUS.1.NAME$
 ",SEQUENTIAL,INTERNAL,INPUT
 ,FIXED
>110 IF EOF(5) THEN 150
>120 INPUT #5:A,B
>130 PRINT A;B
>140 GOTO 110
>150 CLOSE #5
```

INPUT

The EOF function cannot be used with RELATIVE files or with some peripheral devices (cassette recorders). In these cases, you can create your own method for determining if the end-of-file has been reached.

One common end-of-file technique is to create a last record on the file that serves as an end-of-file indicator. This is called a "dummy" record because the data it contains are used only to mark the end of the file. For example, it could be filled with 9's. When the computer inputs a record, you can check if the record is equal to 9's. If so, the computer has reached the end of the file and can skip to a closing routine.

The first example on the right creates a dummy record. In the next example, the computer checks for the dummy record as its end-of-file technique.

```
>NEW

>100 OPEN #2:"CS1",SEQUENTIA
 L,FIXED,OUTPUT,INTERNAL
>110 READ A,B,C
>120 IF A=999999 THEN 180
>130 E=A+B+C
>140 PRINT A;B;C;E
>150 PRINT #2:A,B,C,E
>160 GOTO 110
>170 DATA 5,10,15,10,20,30,10
 0,200,300,99,99,99,
>180 PRINT #2:999999,999999,999999,999999
>190 CLOSE #2
>RUN


  * REWIND CASSETTE TAPE CS1
    THEN PRESS ENTER

  * PRESS CASSETTE RECORD CS1
    THEN PRESS ENTER

 5 10 15 30
 10 20 30 60
 100 200 300 600

  *PRESS CASSETTE STOP CS1
   THEN PRESS ENTER

  ** DONE **


>NEW

>100 OPEN #1:"CS1",INTERNAL,I
 NPUT,FIXED
>110 INPUT #1:A,B,C,E
>120 IF A=99 THEN 160
>130 F=A*E
>140 PRINT A;B;C;E;F
>150 GOTO 110
>160 CLOSE #1
>RUN

  * REWIND CASSETTE TAPE CS1
    THEN PRESS ENTER

  * PRESS CASSETTE PLAY CS1
    THEN PRESS ENTER
   5  10  15  30  150
   10  20  30  60  600
   100  200  300  600  6000

  *PRESS CASSETTE STOP CS1
   THEN PRESS ENTER
 ** DONE **
```

PRINT

PRINT #file-number[,REC numeric-expression][:print-list]

(For a description of the PRINT format for printing on the screen, see the "Input-Output Statements" section.)

This form of the PRINT statement enables you to write data to a peripheral device. The PRINT statement can be used to write only to files opened in OUTPUT, UPDATE, or APPEND mode.

The file-number must be the file-number of a currently open file. See the OPEN statement.

When the computer performs a PRINT statement, it stores the data in a temporary storage area called an input/output (I/O) buffer. A separate buffer is provided for each open file-number. If the PRINT statement does not end with a print-separator (comma, semicolon, or colon), the record is immediately written to the file from the I/O buffer. If the PRINT statement ends with a print-separator, the data are retained in the buffer and a "pending" print condition occurs (see "Using Pending Prints" in this section.)

Using PRINT with INTERNAL Data

The print-list consists of numeric and string expressions separated by commas, colons, or semicolons. All print separators in a print-list have the same effect with INTERNAL data; they only separate the items from each other and do not indicate spacing or character positions in a record.

```
>NEW

>100 OPEN #5:"CS1",SEQUENTIAL
 ,INTERNAL,OUTPUT,FIXED
.
.  (Program lines . . .)
.
>170 PRINT #5:A,B,C$,D$
.
.  (Program lines . . .)
.
>200 CLOSE #5



>NEW

>100 OPEN #6:"CS1",SEQUENTIAL
 ,DISPLAY,OUTPUT,FIXED
.
.  (Program lines . . .)
.
>170 PRINT #6:A;",";B;",";C$;
 ",";D$
.
.  (Program lines . . .)
.
>200 CLOSE #6
```

The print-list items written to a peripheral mass-storage device in INTERNAL format have the following characteristics:

Numeric Items:	(insert the graphic on page II-132) designates length of item (always 8)	value of item
String Items:	(insert the graphic on page II-132) designates length of item	value of item

In the example on the right, the total length of the data recorded in INTERNAL format is 71 positions. Each numeric variable uses 9 positions. A$ (line 110) is 18 characters long plus 1 position to record the length of the string. B$ (line 120) is 15 characters plus 1. If the values of A$ and B$ change during the program, their lengths written to the file are the values present when the record is written.

When designing your records, study the data each variable might contain, and plan your records to allow for the greatest length possible.

For FIXED-length records, the computer pads each INTERNAL record with binary zeros, when necessary, to make each record the specified length.

A record cannot be longer than the length specified in the OPEN statement (or the default length for the device when the record length has not been specified in the OPEN statement). If the data in a print-list exceed the record length, the program stops, and the message FILE ERROR IN line-number appears.

```
>NEW

>100 OPEN #5:"CS1",SEQUENTIAL
 ,INTERNAL,OUTPUT,FIXED 128
>110 A$="TEXAS INSTRUMENTS"
>120 B$="COMPUTER"
>130 READ X,Y,Z
>140 IF X=99 THEN 190
>150 A=X*Y*Z
>160 PRINT #5:A$,X,Y,Z,B$,A
>170 GOTO 130
>180 DATA 5,6,7,1,2,3,10,20,3
,0,20,40,60,1.5,2.3,7.6,99,99
,99
>190 CLOSE #5
>RUN

 *REWIND CASSETTE TAPE  CS1
  THEN PRESS ENTER

 *PRESS CASSETTE RECORD  CS1
  THEN PRESS ENTER

(Data written on tape.)

 *PRESS CASSETTE STOP  CS1
  THEN PRESS ENTER
** DONE **
```

Using_PRINT_with_DISPLAY_Data

Although it is best to store INTERNAL data on mass storage devices, you may occasionally need to use DISPLAY data. There are several important considerations you must observe when using DISPLAY format.

XX ? print-list of screen and how records are set up--are not discussed here?? and are they the same?? Also check that how records are set up is included in INTERNAL section.!!!!!

Records are created according to the specifications found in the PRINT statement of the "Input-Output Statements" section.

If a data item from the print-list causes the record to be longer than the specified record length (or the default length), the item is written as the first item in the next record. If any single item is longer than the record length, the item itself is split into as many records as are required to store it. Normally, the program continues running and no warning is given.

The computer can read DISPLAY data only if they are in the same format as data entered from the keyboard. When you write a record to a file that the computer will later read, you must explicitly include the comma separators and quotation marks needed by the INPUT statement. These punctuation marks are not automatically inserted when the PRINT statement is performed. They must be included as items in the print-list, as shown in line 170 on the right.

Numeric items do not have a fixed length. The length of a numeric item is the same as if it were displayed on the screen by the PRINT or DISPLAY statement (i. e., includes sign, decimal point, exponent, trailing space, etc.). For example, the number of positions required to print 1.35E-10 is ten.

```
>NEW

>100 OPEN #10:"CS1",SEQUENTIA
 L,DISPLAY,OUTPUT,FIXED 128
.
. program lines
.
>170 PRINT #10:"""";A$;"""","
 X;",";Y;",";Z:",""";B$;""","
 ;A
.
. program lines
.
>300 CLOSE #10
```

PRINT

Using PRINT with RELATIVE Files

(See the OPEN statement for a description of RELATIVE file-organization.)

RELATIVE file records can be processed randomly or in sequence. The internal counter points to the next record to be processed. The first record in a file is record 0. Thus, the counter begins at zero and is incremented by +1 after each file access (whether the access read or writes a record). In the example on the right, the PRINT statement directs the computer to write the file sequentially. Note that the file can later be processed either randomly or in sequence.

The internal counter can be changed by using the REC clause. The keyword REC must be followed by a numeric-expression whose value specifies the position to which the record is to be written. When the computer performs a PRINT statement with a REC clause, it begins building an output record in the I/O buffer. When this record is written to the file, it is placed at the location specified by REC. You may use the REC option only with RELATIVE files.

The example on the right illustrates writing records randomly.

Be sure to use the REC clause if you read and write records on the same file within a program. The same internal counter is incremented when records are either read from or written to the same file; if REC is not used, you could skip some records and write over others, as shown in the example on the right.

Note that files written on cassette tape must be accessed in sequential order.

```
>NEW

>100 OPEN #3:"HEXBUS.1.NAME$"
 ,RELATIVE,INTERNAL,OUTPUT,FI
 XED 128
.
.  (Program lines . . .)
.
>150 PRINT #3:A$,B$,C$,X,Y,Z
.
.  (Program lines . . .)
.
>200 CLOSE #3

>NEW

>100 OPEN #3:"HEXBUS.1.NAME$"
 ,RELATIVE,INTERNAL,UPDATE,FI
 XED 128
>110 INPUT K
>120 PRINT #3,REC K:A$,B$,C$,
 X,Y,Z
.
.  (Program lines . . .)
.
>300 CLOSE #3


>NEW

>100 OPEN #3:"HEXBUS.1.NAME$"
 ,RELATIVE,INTERNAL,UPDATE,FI
 XED
>110 FOR K=1 TO 10
>120 INPUT #3:A$,B$,C$,X,Y
>130 PRINT #3:A$,B$,C$,X,Y
>140 NEXT K
>150 CLOSE #3

 (LINE 120 reads records 0,2,4,6,8
 LINE 130 writes records 1,3,5,7,9)
```

PRINT

Using Pending Prints

XX ** are prints discussed earlier?? Do we need thi first sentence??

A record is written to a file when the computer performs a PRINT statement that does not end with a trailing separator.

A pending print occurs when a PRINT statement ends with a trailing print separator. When the next PRINT statement using the file is encountered, one of the following actions occurs:

- If the next PRINT statement has no REC clause, the computer places the data in the I/O buffer immediately following the data already there.

- If the next PRINT statement has a REC clause, the computer writes the pending print record to the file at the position indicated by the internal counter and (****or writes the record at the position specified by REC***) performs the new PRINT statement as usual.

If a pending print condition exists and an INPUT statement for the same file is encountered, the pending print record is written to the file at the position indicated by the internal counter, and the internal counter is incremented. The INPUT statement is then performed. If a pending print condition exists and a CLOSE or RESTORE statement accesses the file, the pending print record is written before the file is closed or restored.

CLOSE

CLOSE #file-number[:DELETE]

The CLOSE statement closes or discontinues the association between a file and a program. After the CLOSE statement is performed, the closed file cannot be accessed by your program because the computer no longer associates the file with the file-number you specified. You can then reassign that particular file-number to any file.

The DELETE option can be used on only certain peripherals. Refer to the peripheral manuals for information on using the DELETE option in a CLOSE statement.

If you attempt to CLOSE a file that is not open, the computer stops the program and displays a FILE ERROR message.

To protect your files, the computer automatically closes any open files when it finds an error that stops a program. When the BREAK command or the BREAK or CLEAR key causes a breakpoint in a program, all open files are closed only if one of the following occurs:

- you edit the program
- you leave TI-99/2 BASIC with the BYE command
- you run the program again
- you enter a NEW command

If you use the QUIT command to leave your program, the computer will NOT close any open files, and you could lose the data on these files. To exit before the normal end of a program that uses files, follow these directions to protect your files:

- Press CLEAR until the computer displays BREAKPOINT AT line-number. (This may take several seconds.)
- Enter BYE when the cursor reappears on the screen.

```
XX REWRITE PROGRAM--2 CASSETTE RECORDERS ARE USED HERE XX

>NEW

>100 OPEN #6:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED
>110 OPEN #25:"CS2",SEQUENTIA
 L,INTERNAL,OUTPUT,FIXED
  .
  . (Program lines . . .)
  .
>200 CLOSE #6:DELETE
>210 CLOSE #25
```

CLOSE

Cassette_Recorder_Information

When the computer performs the CLOSE statement for a cassette tape device, the screen displays instructions for operating the recorder, as shown on the right.

If you use the DELETE option with cassette recorders, no action beyond the closing of the file takes place.

```
> NEW

>100 OPEN #19:"CS1",INTERNAL,
 OUTPUT,FIXED
  .
  . (Program lines . . .)
  .
>210 CLOSE #19
>RUN

* REWIND CASSETTE TAPE    CS1
  THEN PRESS ENTER

* PRESS CASSETTE PLAY     CS1
  THEN PRESS ENTER

  .
  . (Program runs . . .)
  .

* PRESS CASSETTE STOP     CS1
  THEN PRESS ENTER


** DONE **
```

RESTORE

RESTORE #file-number[,REC numeric-expression]

The RESTORE statement repositions an open file to its beginning record (see the first example on the right) or at a specific record if the file is a RELATIVE one (see the second example on the right).

If the file-number specified in a RESTORE statement is not already open, the program stops, and the message FILE ERROR IN line-number appears.

You may use the REC option only with a RELATIVE file. The computer evaluates the numeric-expression following REC and uses the value as a pointer to a specific record on the file. If you restore a RELATIVE file and do not use the REC option, the file is set to record 0.

If there is a pending print record, the record is written to the file before the RESTORE is performed. If there is pending INPUT, the data in the I/O buffer are discarded.

```
>NEW

>100 OPEN #2:"CS1",SEQUENTIAL
 ,INTERNAL,INPUT,FIXED 64
>110 INPUT #2:A,B,C$,D$,X
 .
 .  (Program lines . . .)
 .
>400 RESTORE #2
>410 INPUT #2:A,B,C$,D$,X
 .
 .  (Program lines . . .)
 .
>500 CLOSE #2
```

```
>NEW

>100 OPEN #4:"HEXBUS.1.NAME$"
 ,INTERNAL,UPDATE,FIXED 128
>110 INPUT #4:A,B,C
 .
 .  (Program lines . . .)
 .
>200 PRINT #4:A,B,C
 .
 .  (Program lines . . .)
 .
>300 RESTORE #4,REC 10
>310 INPUT #4:A,B,C
 .
 .  (Program lines . . .)
 .
>400 CLOSE #4
```

SUBPROGRAMS

A subprogram is a predefined sequence of instructions built into BASIC to perform special tasks. Subprograms are accessed in a program line by the keyword CALL followed by the subprogram name. Subprograms can also be accessed in a command.

Information can be passed to a subprogram for it to use and, in return, the subprogram can return information. This information is passed to and received from a subprogram through constants, variables, and/or expressions that are called parameters. Parameters are included in parentheses after the name of the subprogram.

An example of accessing the subprogram HCHAR from a program is shown below.

```
10 CALL HCHAR(12,14,5,1)
```

HCHAR can also be accessed from a command by deleting the line number.

CLEAR subprogram

CALL CLEAR

The CLEAR subprogram is used to clear (erase) the entire screen. When the CLEAR subprogram is called, the space character (code 32) is placed in all positions on the screen.

When the program on the right runs, the screen is cleared before the computer performs the PRINT statements.

```
>PRINT "HELLO THERE!"
 HELLO THERE!
>CALL CLEAR
```

(The screen clears.)

```
>NEW

>100 CALL CLEAR
>110 PRINT "HELLO THERE!"
>120 PRINT "HOW ARE YOU?"
>RUN
```

(The screen clears.)

```
HELLO THERE!
HOW ARE YOU?

** DONE **
```

HCHAR subprogram
(Horizontal Character Repetition)

CALL HCHAR(row=number,column=number,char=code[,number=of=repetitions])

The HCHAR subprogram places a character anywhere on the screen and, optionally, repeats it horizontally. The row=number and column=number locate the starting position on the screen. The row=number, column=number, char=code, and number=of=repetitions are numeric expressions.

If the evaluation of any of the numeric expressions results in a non-integer value, the result is rounded to the nearest integer. The valid ranges are given below:

Value	Range
Row=number	1-24, inclusive
Column=number	1-32, inclusive
Char=code	0-32767, inclusive
Number=of=repetitions	0-32767, inclusive

```
>CALL CLEAR
```

(The screen clears.)

```
>CALL HCHAR(10,1,72,50)
```

(graphic of screen on page II-80

```
XX


>NEW

>100 CALL CLEAR
>110 CHR=40
>120 FOR X=8 TO 22
>130 CALL VCHAR(4,X,CHR,15)
>140 CALL HCHAR(X-4,8,CHR,15)

>150 CHR=CHR+8
>160 NEXT X
>170 GOTO 110
>RUN
```

(The screen clears.)

(The screen displays a pattern.)

(Press BREAK to stop the program.)

HCHAR subprogram

A value of 1 for the row-number indicates the top of the screen. A value of 1 for the column-number indicates the left side of the screen. The screen can be thought of as a "grid" as shown here.

(graphic of screen grid on page II-81)

A character may not appear on your screen in columns 1, 2, 31, or 32; therefore, you may want to use only column-numbers 3 through 30.

Char-code can be from 0 through 32767; however, the computer will repeatedly reduce the value by 256 until it is less than 256. Character codes 32 through 127 are defined as the standard ASCII character codes. Character codes 0 through 31 are defined as special graphics characters.

If you specify a char-code for an undefined character, whatever is in that memory location at that time is displayed.

Number-of-repetitions is the number of times the specified character is repeated. The computer displays the character beginning at the specified starting position and continuing to the right side of the next line. If the bottom of the screen is reached, the display continues on the top line of the screen.

To fill all 24 rows and 32 columns, use 768 for number-of-repetitions. Using a number larger than 768 unnecessarily extends the time required to perform this statement.

```
>CALL HCHAR(24,14,29752)
              8
>CALL HCHAR(24,14,35)
              #
>CALL HCHAR(24,14,132)
```

(The displayed character depends on what is now in memory.)

```
>NEW

>100 CALL CLEAR
>110 FOR K=9 TO 15
>120 CALL HCHAR(K,13,36,6)
>130 NEXT K
>140 GOTO 140
>RUN
```

(The screen clears.)

(graphic of screen on page II-82)

(Press BREAK to stop the program.)

VCHAR subprogram
(Vertical character repetition)

CALL VCHAR(row=number,column=number,char=code[,number=of=repetitions])

The VCHAR subprogram places a character anywhere on the screen and, optionally, repeats it vertically. The row=number and column=number locate the starting position on the screen. The row=number, column=number, char=code, and number=of=repetitions are numeric expressions.

If the evaluation of any of the numeric expressions results in a non-integer value, the result is rounded to the nearest integer. The valid ranges are given below:

Value	Range
Row=number	1-24, inclusive
Column=number	1-32, inclusive
Char=code	0-32767, inclusive
Number=of=repetitions	0-32767, inclusive

VCHAR subprogram

A value of 1 for row-number indicates the top of the screen. A value of 1 for the column-number indicates the left side of the screen. The screen can be thought of as a "grid" as shown here.

(graphic of screen grid on page II-81)

A character might not appear on your screen in columns 1, 2, 31, or 32. This is not due to a faulty television set. Many manufacturers build "overscan" into their picture tubes to compensate for increasingly narrower pictures sometimes found on aging television sets. Therefore, you may want to use only column-numbers 3 through 30.

Char-code can be from 0 through 32767; however, the computer will repeatedly reduce the value by 256 until it is less than 256. Character codes 32 through 127 are defined as the standard ASCII character codes. Character codes 0 through 31 are defined as special graphics characters.

If you specify a char-code for an undefined character, whatever is in that memory location at that time is displayed..

Number-of-repetitions is the number of times the specified character is repeated. The computer displays the character beginning at the specified starting position and continuing down the screen. If the bottom of the screen is reached, the display continues at the top of the next column to the right. If the right edge of the screen is reached, the display continues at the left edge.

To fill all 24 rows and 32 columns, use 768 for number-of-repetitions. Using a number larger than 768 unnecessarily extends the time required to perform this statement.

```
>CALL CLEAR

  (The screen clears.)

>CALL VCHAR(2,10,86,13)
```

(graphic of first screen on page II-83)

```
>NEW

>100 CALL CLEAR
>110 FOR K=13 TO 18
>120 CALL VCHAR(9,K,36,6)
>130 NEXT K
>140 GOTO 140
>RUN
```

(The screen clears.)

(graphic of second screen on page II-83)

(Press BREAK to stop the program.)

GCHAR subprogram
(Get character)

CALL GCHAR(row-number,column-number,numeric-variable)

The GCHAR subprogram enables you to read the character that is located at any position on the screen. The position of the character is described by row-number and column-number. The ASCII numeric code of the requested character is stored in the numeric-variable you specify in the CALL GCHAR statement.

The row-number and column-number are numeric expressions. If the evaluation of either numeric expression results in a non-integer value, the value is rounded to the nearest integer. A value of 1 for row-number indicates the top of the screen. A value of 1 for column-number specifies the left side of the screen. The screen can be thought of as a "grid" as shown here.

(graphic of screen grid on page II-86)

```
>NEW

>100 CALL CLEAR
>110 CALL HCHAR(1,1,36,768)
>120 CALL GCHAR(5,10,X)
>130 CALL CLEAR
>140 PRINT X
>RUN

  (The screen clears.)

  (The screen fills with $'s, code 36.)

  (The screen clears.)

  36

 ** DONE **
```

KEY subprogram

CALL KEY(key-unit,return-variable,status-variable)

The KEY subprogram enables you to determine when a key on the console is pressed. The character corresponding to the pressed key is input to your program. CALL KEY eliminates the need for an INPUT statement, and because the character represented by the key pressed is not displayed on the screen, the information already on the screen is not disturbed.

The key-unit is a numeric expression that must have a value of zero, which indicates that the keyboard is the input device.

The return-variable is the numeric variable where the computer places the numeric character code corresponding to the key(s) pressed. The numeric character code is a number from 0 through 127. Refer to appendix XX for a list of the character codes.

The status-variable is a numeric variable. The computer places in status-variable one of the following codes:

!o! +1 means a new key was pressed since the last CALL KEY was performed.

!o! -1 means the same key was pressed as was returned in the last CALL KEY.

!o! 0 means no key was pressed.

```
>NEW

>100 CALL CLEAR
>110 FOR R=1 TO 24
>120 FOR C=3 TO 30
>130 CALL HCHAR(R,C,17)
>140 CALL KEY(0,K,S)
>150 IF K<>32 THEN 140
>160 NEXT C
>170 NEXT R
>RUN
```

(Solid box appears at row 1, column 3 of screen. Press SPACE BAR to display more boxes. When screen is filled, the program ends. Press BREAK to end program earlier.)

PEEK subprogram

CALL PEEK(address,numeric-variable1[,numeric-variable2,...])

The PEEK subprogram is used to read the contents of memory locations. Starting at the memory location specified by address, the value of that byte of memory is assigned to numeric-variable1, the value of the next byte to numeric-variable2, and so forth. The number of variables listed after the address determines how many bytes are read.

Address must be a numeric expression from 0 to 65535. The values assigned to the variables are in the range 0 through 255.

POKE subprogram

CALL POKE(address,byte1[,byte2,...])

The POKE subprogram is used to write data into memory locations. The value of byte1 is stored in the memory location specified by address, the value of byte2 is stored in the next memory location, and so forth.

The value of each data byte can be from 0 through 255. If the value is greater than 255, it is repeatedly reduced by 256 until it is less than 256. Using a byte value greater than 32767 causes an error.

Indiscriminate use of this statement may destroy the program currently in memory and require that the computer be reset to continue.

```
>100 CALL PEEK(2096,X1,X2,X3,
 X4)
```

Returns the values in locations 2096, 2097, 2098, and 2099 and assigns them to variables X1, X2, X3, and X4, respectively.

```
>100 CALL POKE(850,162,10,17)
```

Places the values 162, 10, and 17 in the locations 850, 851, and 852, respectively.

MaCHine Language subprogram

CALL MCHL(address)

This subprogram enables experienced programmers to run assembly language programs. For more information on the assembly language used by the TMS9995 microprocessor, refer to the appropriate manuals.

Before using this command, you must reserve an area of memory for the assembly language progam and enter the program. The area you reserve should be immediately below the memory used for your BASIC program. The bottom of this memory is at the location specified at address -4086. You reserve the memory by finding the value at that address, adding the number of bytes your assembly language program needs, and putting the new value back into the address.

The following program segment reserves an area of memory for an assembly language program, assuming that the number of bytes of memory needed has been previously assigned to the variable MEMRQD.

```
>140 CALL PEEK(-4086,HEX1,HEX2)
>150 DECIMAL=HEX1*256+HEX2
>160 NEWDECIMAL=DECIMAL+MEMRQD
>170 HEXNEW1=INT(NEWDECIMAL/256)
>180 HEXNEW2=NEWDECIMAL-HEXNEW1*256
>190 CALL POKE(____,HEXNEW1,HEXNEW2)
```

After the area has been reserved, put your assembly language program into memory using POKE statements, starting at the address that was previously the bottom of memory. Assembly language programs should start on an even-numbered address byte.

```
>200 DECIMAL=INT((DECIMAL+1)/2)*2
>210 CALL POKE(DECIMAL+1,value)
>220 CALL POKE(DECIMAL+2,value)
.
. (Program lines . . .)
.
>290 CALL POKE (DECIMAL+MEMRQD,value)
```

To run the assembly language program, enter CALL MCHL(address) as a program line, where address is the entry point of the program.

```
>300 CALL MCHL(address)
```

The computer executes the routine. When the assembly language program ends, assuming the machine language program does not alter any pointers or other items used by BASIC, program execution resumes at the line after CALL MCHL.

GLOSSARY

Accessory_Devices--See Peripheral_Devices.

Array--A collection of numeric or string variables arranged in a list or matrix for processing by the computer. Each element in an array is referenced by a subscript describing its position in the list.

ASCII--The American Standard Code for Information Interchange, the code structure used internally in most personal computers to represent letters, numbers, and special characters.

BASIC--(Beginners All-purpose Symbolic Instruction Code)--An easy-to-use popular programming language used in most personal computers. It was developed at Dartmouth College in the 60's.

Baud--The transmission rate, in bits per second, of data over a communication line, such as between a computer and a peripheral. 300 baud indicates approximately 300 bits of information are being transmitted serially every second.

Binary--The two-digit (bit) number system based on 0 and 1. Computers recognize the binary bits 0 and 1 by using gates. Gates are electronic circuits that are either off or on, representing 0 or 1, respectively.

Bit--A binary digit (0 or 1).

Branch--A departure from the sequential performance of program statements. An unconditional branch causes the computer to jump to a specified program line every time the branching statement is encountered. A conditional branch transfers program control contingent on the result of some arithmetic or logical operation.

Breakpoint--A point in a program specified by the BREAK command at which program execution is suspended. During a breakpoint, you can perform operations in the Immediate_Mode to help you locate program errors. Program execution can be resumed with a CONTINUE command, unless the program was edited during the break.

Buffer--An area of computer memory used for temporary storage of an input or output record.

Bug--An error in the hardware or software of a computer that causes an intended operation to be performed incorrectly.

Byte--A string of binary digits (bits) treated as a unit, often representing one data character. The computer's memory capacity is often expressed as the number of bytes available. For example, a computer with "16K" has about 16,000 bytes of memory available for storing programs and data.

Cassette--A standard audio cassette tape that is used to store programs and other data; the same type of tape commonly used to record music. (Use of "metal" tapes is not recommended.)

Central Processing Unit (CPU)--The nerve center of a computer; the network of electronic circuits that interprets programs and tells a computer how to carry them out.

Character--A letter, number, punctuation symbol, or special graphics symbol, usually equivalent to one byte.

Chip--Tiny silicon slices used to make electronic memories and other circuits. A single chip may have as many as 30,000 electronic parts.

Circuit Board--A rigid fiberglass or phenolic card on which various electronic parts are mounted. Printed or etched copper tracks connect the various parts to one another.

Command--An instruction that the computer performs immediately. Commands are not a part of a program and thus are entered with no preceding line number.
Examples: NEW, LIST, RUN, CALL CLEAR.

Computer--A network of electronic switches and memories that processes data.

Concatenation--The linking of two or more strings to make a longer string. The "&" is the concatenation operator.

Constant--A numeric real number (such as 1.2 or -9054) or a string of characters (any combination of up to 112 characters enclosed in quotes, such as"HELLO THERE" or "275 FIRST ST.")

CPU--See Central Processing Unit.

Cursor--A flashing underline or rectangle that indicates where a typed character appears.

Data--Basic elements of information that are processed or produced by the computer. The singular form seldom used is datum.

Default--A standard characteristic or value that the computer assumes if certain specifications are omitted within a statement or program.

Display--As a noun, the video screen; as a verb, to cause characters to appear on the screen.

Edit Mode--The mode used to change existing program lines. The EDIT mode is entered by using the Edit Command or by entering the line number followed by SHIFT E (UP ARROW KEY) or SHIFT X (DOWN ARROW KEY). The line specified is displayed on the screen and changes can be made to any character (except the line-number) using the editing keys.

End-of-file (EOF)--The condition indicating that all data have been read from a file.

Execute--To run a program; to perform the task specified by a statement or command.

Exponent--A number indicating the power to which a number or expression is to be raised, usually written to the right and above the number.

For example: 2^8 =2x2x2x2x2x2x2x2. In TI-99/2 BASIC, the exponent is entered following the ^ symbol or following the letter "E" in scientific notation.

For example: 2^8 =2^8; 1.3×10^{25} =1.3E25 (or 1.3E+25).

Exponential Notation--See scientific notation.

Expression--A combination of constants, variables, and operators that can be evaluated to a single result; expressions can be numeric, string, relational, or logical.

File--A collection of related data records stored on a peripheral device; also used interchangeably with "device" for input/output equipment that cannot use multiple files, such as a line printer.

Fixed-length records--File records that are all the same length. If a file has fixed-length records of 95 characters, each record is allocated 95 bytes even if the data occupy only 76 positions. The computer adds padding characters on the right to ensure that the record has the specified length.

Function--A feature that enables you to specify as "single" operations a variety of procedures, each of which actually contains a number of steps; for example, a procedure to calculate square roots via a simple reference name.

Gate--A very simple electronic circuit that is always either on or off. Clusters of gates can manipulate binary numbers (0 = off, 1 = on). They can also count, do arithmetic, make decisions, and store binary numbers. Gates are the basic building blocks of computers.

Hardware--The various devices that comprise a computer system, including memory, the keyboard, the screen, data storage/retrieval devices, line printers, etc.

Hertz--A unit of frequency; one Hertz = one cycle per second.

Hexadecimal--A base 16 number system using 16 symbols, 0-9 and A-F. It is used as a convenient "shorthand" way to express binary code; for example, 1010 in binary is A in hexadecimal; 11111111 in binary is FF in hexadecimal.

Immediate Mode--A computer mode in which commands are entered directly into the computer without a line number; such commands are executed immediately. Also called Command Mode.

Increment--A positive or negative value that is used to modify a variable.

Input--As a noun, data entered into memory to be processed; as a verb, the process of transferring data into memory.

Input line--The amount of data that can be entered at one time; in TI-99/2 BASIC, 112 characters.

Internal_data_format--Data in the form used directly by the computer. Internal numeric data are 8 bytes long plus 1 byte that specifies the length; the length for internal string data is one byte per character in the string plus one length-byte.

Integer--A whole number, either positive, negative, or zero.

Interpreter--The program stored inside a computer that converts or translates TI-99/2 BASIC statements into the computer's machine language.

Input/Output_(I/O)--Usually refers to a device function; I/O is used for communication between the computer and other devices (e.g., keyboard, Program Recorder).

Iteration--The technique of repeating a group of program statements; one repetition of such a group. See Loop.

K--Short for "kilo," meaning "thousand"; 1K of memory is actually 1024 (2^{10}) bytes. Thus, a 4K memory has approximately 4,000 storage elements.

Line--See input line, print line, or program line.

Line_number--A number identifying a statement in a program; line numbers determine the order in which a computer executes the commands of a program.

Loop--A group of consecutive program lines repeatedly performed, usually a specified number of times.

Mantissa--The base-number portion of a number expressed in scientific notation; in 3.264E+4, the mantissa is 3.264.

Mass-storage_device--A peripheral_device (such as the Program Recorder or Wafertape™ Drive) that stores programs or data for later use by the computer.

Memory--See RAM, ROM, and mass storage device.

Microprocessor--The central processing unit of a computer assembled on a single silicon chip.

Null_string--A string that contains no characters and has zero length.

Number_mode--The mode in which the computer automatically generates program line numbers for entering or changing statements.

Operator--A symbol used in calculations (arithmetic operators), in comparisons (relational operators), and string concatenation (linkage). The arithmetic operators are +, -, *, /, and ^. The relational operators are >, <, =, >=, <=, and <>. The string operator is &.

Output--As a noun, information supplied by the computer; as a verb, the process of transferring information from the computer's memory to a peripheral device, such as a screen, printer, or mass-storage_device.

Overflow--The condition that occurs when a rounded value greater than 9.9999999999999E127 or less than -9.9999999999999E127 is entered or computed. When an overflow occurs, the value is replaced by the computer's limit, a warning is displayed, and the program continues.

Parameter--A value that affects the output of a statement or function.

Peripheral_Devices--Equipment that attaches to the computer to extend its functions and capabilities; these units send, receive, or store data. They are often called simply peripherals.

Print_line--A 28-position line used by the PRINT and DISPLAY statements.

Program--A set of statements that tells the computer how to perform a complete task.

Program_line- A line containing a single statement, the maximum length of which is 112 characters.

Prompt--A symbol (>) that marks the beginning of each command or program line; a symbol or phrase that requests input from the user.

Pseudo-random_number--A number produced by a set of calculations (an algorithm), sufficiently random for most applications. A truly random number is obtained entirely by chance.

Radix-100--A number system based on 100; see "Accuracy Information."

RAM--Random-access memory; the memory where program statements and data are stored during program execution. New programs and data can be read in, accessed, and changed in RAM. Data stored in RAM are erased when the power is turned off or BASIC is exited.

Record--A collection of related data, such as an individual's payroll information or a student's test scores; a group of similar records, such as a company's payroll records, is called a file.

Reserved_word--A special word with a predefined meaning in programming languages. A reserved word must be spelled precisely, appear in its proper position in a statement or command, have one space preceding and following it, and must not be used as a variable name.

ROM--Read-only memory; the memory where certain instructions for the computer are permanently stored; ROM can be read but cannot be changed. ROM is not erased when electrical power is turned off.

Run_Mode--The mode in which the computer executes a program. Run Mode is terminated when program execution ends, either normally or abnormally. To leave Run Mode, press CLEAR during program execution (see Breakpoint).

Scientific_Notation--A method of expressing very large or very small numbers by using a base number (mantissa) times 10 raised to some power (exponent). To represent scientific notation in TI-99/2 BASIC, enter the mantissa (preceded by the minus sign if negative), the letter E, and the power of 10 (preceded by a minus sign if negative). For example, 3.264E4; -2.47E-17. This special format of scientific notation is called exponential notation.

Scroll--Movement of text on the screen to display additional information.

Software--Programs that are executed by the computer, including programs built into the computer, programs on cassettes or wafers, and programs entered by the user.

Statement--An instruction (preceded by a line number) in a program. In TI-99/2 BASIC, only one statement is allowed in a program_line.

String--A series of letters, numbers, and symbols treated as a unit.

Subprogram--A predefined, general-purpose procedure accessible to the user through the CALL statement in TI-99/2 BASIC. Subprograms extend the capability of BASIC.

Subroutine--A program segment that can be used more than once during the execution of a program to perform a special task (e.g., a set of calculations or a print routine). In TI-99/2 BASIC, a subroutine is accessed by a GOSUB statement and terminated with a RETURN statement.

Subscript--A numeric expression that specifies a particular item in an array; in TI-99/2 BASIC, the subscript is written in parentheses immediately following the array name.

Trace--A command that lists the order in which the computer performs program statements; tracing line numbers can help you find errors in a program.

Underflow--The condition that occurs when the computer generates a numeric value greater than -1E-128, less than 1E-128, and not zero. When an underflow occurs, the value is replaced by zero.

Variable--A value that may vary during program execution. A variable is stored in a memory location and can be replaced by new values during program execution.

Variable-length_records--Records in a file that vary in length depending on the amount of data per record. Using variable-length records conserves space on a file. Variable-length records must be accessed sequentially.

APPENDIX XX ASCII CHARACTER CODES

The following is a list of the ASCII character codes in decimal notation and their corresponding characters. Graphics symbols are assigned to codes 0 through 31. The Basic Computer 99/2 uses standard ASCII characters for codes 32 through 127. The cursor is assigned to code XX and the edge character is assigned to code XX.

Note that the characters corresponding to codes 98 through 127 cannot be displayed when entered from the keyboard. You may, however, display them with either HCHAR or VCHAR.

ASCII CODE	CHARACTER
00	GS
01	GS
02	GS
03	GS
04	GS
05	GS
06	GS
07	GS
08	GS
09	GS
10	GS
11	GS
12	GS
13	GS
14	GS
15	GS
16	GS
17	GS
18	GS
19	GS
20	GS
21	GS
22	GS
23	GS
24	GS
25	GS
26	GS
27	GS,
28	GS
29	GS
30	GS
31	GS
32	(space)
33	! (exclamation point)
34	" (quote)
35	# (number or pound sign)
36	$ (dollar)
37	% (percent)
38	& (ampersand)
39	' (apostrophe)

40	((open parenthesis)
41	) (close parenthesis)
42	* (asterisk)
43	+ (plus)
44	, (comma)
45	- (minus)
46	. (period)
47	/ (slant)
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	: (colon)
59	; (semicolon)
60	< (less than)
61	= (equals)
62	> (greater than)
63	? (question mark)
64	@ (at)
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y

```
 90   Z
 91   [ (open bracket)
 92     (reverse slant)
 93   ] (close bracket)
 94   ^ (caret)
 95   _ (underline)
 96     (grave)
 97  a
 98  b
 99  c
100 d
101 e
102 f
103 g
104 h
105 i
106 j
107 k
108 l
109 m
110 n
111 o
112 p
113 q
114 r
115 s
116 t
117 u
118 v
119 w
120 x
121 y
122 z
123 § (left brace)
124   (vertical line)
125 § (right brace)
126   (tilde)
127 DEL (appears on screen as a blank)
```

FUNCTION KEY CODES

The function keys are assigned the following codes. These codes are returned by the CALL KEY subprogram when the corresponding keys are pressed.

KEY CODE	Function Name	Function Key
1	AID	ECIN Z
2	CLEAR	ECIN 4
3	DELete	ECIN 1
4	INSert	ECIN 2
5	QUIT	ECIN =
6	REDO	ECIN 8
7	ERASE	ECIN 3
8	LEFT ARROW	ECIN S
9	RIGHT ARROW	ECIN D
10	DOWN ARROW	ECIN X
11	UP ARROW	ECIN E
12	PROC'D	ECIN 6
13	ENTER	ENTER
14	BEGIN	ECIN 5
15	BACK	ECIN 9

ERROR MESSAGES

1. ERRORS FOUND WHEN ENTERING A LINE

* BAD LINE NUMBER
  !o! Line number or line number referenced is less than 1 or greater than 32767.
  !o! RESEQUENCE specifications generate a line number greater than 32767.

* BAD NAME
  !o! The variable name has more than 15 characters.

* CAN'T CONTINUE
  !o! CONTINUE was entered with no previous breakpoint or program was edited after a breakpoint occurred.

* CAN'T DO THAT
  !o! Attempted use of the following program statements as commands: DATA, DEF, FOR, GOTO, GOSUB, IF, INPUT, NEXT, ON, OPTION, RETURN.
  !o! Attempted use of the following commands as program statements (entered with a line number): BYE, CONTINUE, EDIT, LIST, NEW, NUMBER, OLD, RUN, SAVE.
  !o! Entering LIST, RUN, or SAVE with no program in memory.

* INCORRECT STATEMENT
  !o! Two variable names in a row with no valid separator between them (A$B).
  !o! A numeric constant immediately following a variable with no valid separator between them (N 257).
  !o! A quoted string with no closing quote mark.
  !o! Invalid print separator between numbers in the LIST, NUMBER, or RESEQUENCE commands.
  !o! Invalid characters following CONTINUE, LIST, NUMBER, RESEQUENCE, or RUN commands.
  !o! Command keyword not the first word in a line.
  !o! Colon not following the device name in a LIST command.

* LINE TOO LONG
  !o! Input line too long for the input buffer.

* MEMORY FULL
  !o! Entering an edit line that exceeds available memory.
  !o! A line added to a program that causes the program to exceed available memory.

II. ERRORS FOUND WHEN SYMBOL TABLE IS GENERATED

After RUN is entered but before any program lines are performed, the computer scans the program to establish a symbol table. A symbol table is an area of memory where the variables, arrays, functions, etc. for a program are stored. A program does not run until its symbol table is generated.

During the scanning process, the computer recognizes certain types of errors. If an error is detected during the scanning process, no program lines are performed and all the values in the symbol table are zero (for numbers) or null (for strings).

The error message displayed contains the line number of the statement which caused the error. The message BAD VALUE IN 100 informs you that line 100 contains a bad value. The error list below contains more information on the cause of the errors.

* BAD VALUE
  !o! A dimension for an array greater than 32767.
  !o! A dimension for an array of 0 when OPTION BASE = 1.

* CAN'T DO THAT
  !o! More than one OPTION BASE statement in your program.
  !o! The OPTION BASE statement with a higher line number than an array definition.

* FOR-NEXT ERROR
  !o! Mismatching of FOR and NEXT statements.

* INCORRECT STATEMENT
  DEF
    !o! No closing parenthesis ")" after a parameter in a DEF statement.
    !o! Equals sign (=) missing in DEF statement.
    !o! Invalid variable name for parameter in DEF statement.

  DIM
    !o! DIM statement with no dimensions or with more than three dimensions.
    !o! A dimension in a DIM statement not a number.
    !o! A dimension in a DIM statement not followed by a comma or a closing ")".
    !o! The array-name in a DIM statement not a valid variable name.
    !o! The closing ")" missing for array subscripts.

  OPTION BASE
    !o! OPTION not followed by BASE.
    !o! OPTION BASE not followed by 0 or 1.

* MEMORY FULL
  !o! Array size too large.
  !o! Not enough memory to allocate a variable or function.

* NAME CONFLICT

!o! The same name assigned to more than one array, e.g., DIM A(5), A(2,7).
!o! The same name assigned to an array and a simple variable.
!o! The same name assigned to a variable and a function.
!o! References to an array have a different number of dimensions for the array, e.g., B=A(2,7)+2,PRINT A(5).

III.ERRORS FOUND WHEN A PROGRAM IS RUNNING

When a program is running and the computer encounters a statement that it cannot perform, an error message is printed. The program terminates unless the error is only a warning. All variables in the program have the values assigned when the error occurred. The number of the line containing the error is printed with the error message (for example, CAN'T DO THAT IN 210).

* BAD ARGUMENT
- !o! A built-in function with a bad argument.
- !o! String expression for either ASC or VAL with a zero length (null string).
- !o! String expression in VAL not a valid representation of a numeric constant.

* BAD LINE NUMBER
- !o! Specified line number in branching statement (GOTO, GOSUB, IF THEN, IF THEN ELSE, ON GOTO, ON GOSUB) nonexistent.
- !o! Specified line number in BREAK or UNBREAK nonexistent (warning only).

* BAD NAME
- !o! Invalid subprogram name in a CALL statement.

* BAD SUBSCRIPT
- !o! Subscript not an integer.
- !o! Subscript with a value greater than the specified or allowed dimensions of an array.
- !o! Subscript 0 used when OPTION BASE 1 specified.

* BAD VALUE

CHAR
- !o! Character-code in CHAR statement out of range.
- !o! Invalid character in pattern-identifier in CHAR statement.

CHR$
- !o! Argument in CHR$ negative or larger than 32767.

EXPONENTIATION (^)
- !o! Attempt to raise a negative number to a fractional power.

FOR
- !o! Step increment of zero in FOR TO STEP statement.

HCHAR, VCHAR, or GCHAR
- !o! Row or column number in HCHAR, VCHAR, or GCHAR statement out of range.

KEY
- !o! Key unit in KEY statement out of range.

ON
- !o! Numeric expression indexing a line number out of range.

OPEN, CLOSE, INPUT, PRINT, RESTORE
- !o! File number negative or greater than 255.
- !o! Number of records in the SEQUENTIAL option of an OPEN statement non-numeric or greater than 32767.
- !o! Record-length greater than 32767 in the FIXED option of an OPEN statement.

POS
- !o! Numeric expression that is negative, zero, or larger than 32767 in the POS statement.

SEG$
- !o! The value of numeric-expression1 (character position) or numeric-expression2 (length of substring) negative or larger than 32767.

TAB
- !o! The value of the character position in the TAB function greater than 32767.

* CAN'T DO THAT
- !o! RETURN with no previous GOSUB statement.
- !o! NEXT with no previous matching FOR statement.
- !o! The control-variable in a NEXT statement not matched with the control-variable in the previous FOR statement.
- !o! BREAK command with no line number.

* DATA ERROR
- !o! No comma between items in DATA statement.
- !o! Variable-list in READ statement not filled but no more DATA statements available.
- !o! READ statement with no DATA statement remaining.
- !o! Assigned a string value to a numeric variable in a READ statement.
- !o! Line-number in RESTORE statement greater than the highest line number in the program.

* FILE ERROR
- !o! Attempt to CLOSE, INPUT, PRINT, or RESTORE a file not currently open.
- !o! Attempt to INPUT records from a file opened in OUTPUT or APPEND mode.
- !o! Attempt to PRINT records to a file opened in INPUT mode.
- !o! Attempt to OPEN a file that is already open.

* INCORRECT STATEMENT

General
- !o! Open parenthesis, "(", close parenthesis, ")", or both missing.
- !o! Comma missing.
- !o! No line number where expected in a BREAK, UNBREAK, or RESTORE (BREAK 100,).
- !o! "+" or "-" not followed by a numeric expression.
- !o! Expressions used with arithmetic operators not numeric.
- !o! Expressions used with relational operators not the same type.
- !o! Attempt to use a string expression as a subscript.
- !o! Attempt to assign a value to a function.
- !o! Reserved word out of order.
- !o! Unexpected arithmetic or relational operator present.
- !o! Expected arithmetic or relational operator missing.

Built-in Subprograms

!o! The key-status in KEY not a numeric variable.
!o! The third specification in GCHAR not a numeric variable.
!o! CALL not followed by a subprogram name.

File Processing-Input/Output Statements

!o! Missing number sign (#) or colon (:) in file-number specification for OPEN, CLOSE, INPUT, PRINT, or RESTORE.
!o! Filename in OPEN or DELETE not a string expression.
!o! Keyword in OPEN statement that is invalid or appears more than once.
!o! The number of records less than zero or greater than 255 in the SEQUENTIAL option of an OPEN statement.
!o! A record length of less than zero or greater than 255 in the FIXED option of an OPEN statement.
!o! A colon (:) in the CLOSE statement not followed by the keyword DELETE.
!o! Required print separator (comma, semicolon, colon) missing in the PRINT statement.
!o! Filename in SAVE or OLD command not a valid string expression.

General Program Statements

FOR

!o! The keyword FOR not followed by a numeric variable.
!o! The control-variable in a FOR statement not followed by an equals sign (=).
!o! The keyword TO missing in a FOR statement.
!o! The limit in a FOR statement not followed by the end of line or the keyword STEP.

IF
!o! The keyword THEN missing or not followed by a line number.

LET
!o! Equals sign (=) missing.

NEXT
!o! The keyword NEXT not followed by control variable.

ON-GOTO, ON-GOSUB
!o! ON not followed by a valid numeric expression.

RETURN
!o! RETURN followed by an unexpected word or character.

User-Defined Functions
!o! Mismatch between the number of function arguments and the number of parameters for a user-defined function.

* INPUT ERROR
!o! Input data too long for Input/Output buffer (only a warning when data are entered from the keyboard; data can be re-entered).
!o! Mismatch between number of variables in the variable-list and number of data items input from keyboard or data file (only a warning if from keyboard).
!o! Non-numeric data INPUT for a numeric variable (this condition could be caused by reading padded characters on a file record; only a warning if from keyboard).
!o! Numeric INPUT data causing an overflow (only a warning if from keyboard).

* I/O ERROR--This condition generates an accompanying error code as follows:

A two-digit error code (XY) is displayed with the message

* I/O ERROR XY IN line-number

where the first digit (X) indicates which I/O operation caused the error and the second digit (Y) indicates the kind of error that occurred as shown below.

X Value	Operation
0	OPEN
1	CLOSE
2	INPUT
3	PRINT
4	RESTORE
5	OLD
6	SAVE
7	DELETE

Y Value Error Type

0 Device name not found (invalid device or file name in DELETE, LIST, OLD, or SAVE command).
1 Device write-protected (attempted to write to a protected file).
2 Bad OPEN attribute (one or more OPEN options were illegal or did not match the file characteristics).
3 Illegal operation (input/output command not valid).
4 Out of space (attempt to write with insufficient space remaining on the storage medium).
5 End of file (attempted to read past the end of a file).
6 Device error (device damaged or not connected; this error can occur during file processing if a peripheral device is accidentally disconnected while the program is running).
7 File error (the indicated file does not exist or the file type does not match).

* MEMORY FULL
  !o! GOSUB statement branching to its own line-number.
  !o! Too many pending subroutine branches with no RETURN performed.
  !o! Too many user-defined functions that refer to other user-defined functions.
  !o! Relational, string, or numeric expression too long.
  !o! User-defined function referencing itself.

* NUMBER TOO BIG (only a warning; value is replaced by the computer limit as shown below).
  !o! A numeric operation producing an overflow (value greater than 9.999999999999E127 or less than -9.999999999999E127).
  !o! A READ statement attempt to assign an overflow value to a numeric variable.
  !o! An INPUT statement attempt to assign an overflow value to a numeric variable.

* STRING-NUMBER MISMATCH
  !o! A non-numeric argument specified for a built-in function, TAB function, or exponentiation operation.
  !o! A non-numeric value in a specification requiring a numeric value.
  !o! A non-string value in a specification requiring a string value.
  !o! Function argument and parameter for a user-defined function disagree in type.
  !o! File-number in OPEN, CLOSE, INPUT, PRINT, or RESTORE not numeric.
  !o! Attempt to assign a string to a numeric variable.
  !o! Attempt to assign a number to a string variable.

Note: Additional error codes may occur when you use peripherals, such as the HEX-BUS™ devices. Consult the appropriate peripheral owner's manual for more information on these error codes.

IV. ERROR RETURNED WHEN AN OLD COMMAND IS NOT SUCCESSFUL

* CHECK PROGRAM IN MEMORY
  The OLD command does not clear program memory unless the loading operation is successful. If an OLD command fails or is interrupted, however, any program currently in memory may be partially or completely overwritten by the program being loaded. LIST the program in memory before proceeding.

ACCURACY INFORMATION

Displayed_Results_Versus_Accuracy

The Basic Computer 99/2, like all other computers, operates under a fixed set of rules within preset limits.

The mathematical tolerance of the computer is controlled by the number of digits it uses for calculations. The computer appears to use 10 digits as shown by the display, but actually uses more to perform all calculations. When rounded for display purposes, these extra digits help maintain the accuracy of the values presented. Example:

```
1/3X3 = .9999999999 (inaccurate)
```

The example shows that 1/3 = .3333333333, when multiplied by 3, produces an inaccurate answer. However, a 13-digit string of nines, when rounded to 10 places, is 1.0000000000.

The higher-order mathematical functions use iterative and polynomial calculations. The cumulative rounding error is usually kept beyond the tenth digit so that displayed values are accurate.

Normally, there is no need to consider the undisplayed digits. With certain calculations, however, these digits may appear as an answer when not expected. The mathematical limits of a finite operation (word length, truncation, and rounding errors) do not allow these digits to be always completely accurate. Therefore, when subtracting two expressions that are mathematically equal, the computer may display a nonzero result.

Example:

```
X=2/3-1/3-1/3
PRINT X
1E-14
```

The final result indicates a discrepancy in the fourteenth digit.

Such possible discrepancies in the least significant digits of a calculated result are important when testing if a calculated result is equal to another value. For the previous example, the statement shown below can be used to truncate the undisplayed digits of the variable X, leaving only the rounded display value.

```
X=1E-10*(INT(X*1E10))
```

Internal_Numeric_Representation

The TI-99/2 Computer uses radix-100 format for internal calculations. A single radix-100 digit has a range of value from 0 to 99 in base 10.

The computer uses a 7-digit mantissa, which results in 13 to 14 digits of decimal precision. A radix-100 exponent ranges in value from -64 to +63, which yield decimal exponents from 10^{-128} to 10^{+126}. The exponent and the 7-digit mantissa combine to provide a decimal range from -9.99999999999999E127 through -1.00000000000000E-128; zero; and then +1.00000000000000E-128 through +9.99999999999999E127.

The internal representation of the radix 100 format requires eight bytes. The first byte contains the exponent and the algebraic sign of the entire floating-point number. The exponent is a 7-bit hexadecimal value offset or biased by 40_{16} (the 16 subscript indicates hexadecimal values in this appendix). The correspondence between exponent values is shown below.

Biased hexadecimal value	00_{16}	to	40_{16}	to	$7F_{16}$
Radix-100 value	-64	to	0	to	+63
Decimal value	-128	to	0	to	126

If the floating-point number is negative, the first byte (the exponent value) is inverted (1's complement). Each byte of the mantissa contains a radix-100 digit from 0 to 99 represented in binary coded decimal (BCD) form. In other words, the most significant four bits of each byte represent a decimal digit from 0 to 9 and the least significant four bits represent a decimal digit from 0 to 9. The first byte of the mantissa contains the most significant digit of the radix-100 number. The number is normalized so that the decimal point immediately follows the most significant radix-100 digit.

The following examples shown some decimal values and their internal representations.

Decimal Number	Internal Value							
127 10	41	01	18	00	00	00	00	00
0.5 10	3F	32	00	00	00	00	00	00
Ü/2	40	01	39	07	60	20	43	5F
-Ü/2	BF	FF	39	07	60	20	43	5F

RESERVED WORDS

The following is a complete list of all reserved word in TI-99/2 BASIC. Reserved words are words that are reserved for use by TI-99/2 BASIC and may not be used as variable names. However, you may use a reserved word as part of a variable name (for example, ALEN and LENGTH are allowed).

ABS
APPEND
ASC
ATN
BASE
BREAK
BYE
CALL
CHR$
CLEAR
CLOSE
CON
CONTINUE
COS
DATA
DEF
DELETE
DIM
DISPLAY
EDIT
ELSE
END
EOF
EXP
FIXED
FOR
GCHAR
GO
GOSUB
GOTO
HCHAR
IF
INPUT
INT
INTERNAL
KEY
LEN
LET
LIST
LOG
NEW
NEXT
NUM
NUMBER

OLD
ON
OPEN
OPTION
OUTPUT
PEEK
PERMANENT
POKE
POS
PRINT
RANDOMIZE
READ
REC
RELATIVE
REM
RES
RESEQUENCE
RESTORE
RETURN
RND
RUN
SAVE
SEG$
SEQUENTIAL
SGN
SIN
SQR
STEP
STOP
STR$
SUB
TAB
TAN
THEN
TO
TRACE
UNBREAK
UNTRACE
UPDATE
VAL
VARIABLE
VCHAR

Blank Spaces

In general, a blank space can occur almost anywhere in a program without affecting the execuation of the program. However, any extra blank spaces you put in that are not required will be deleted when the program line is displayed by the EDIT, NUM, or LIST command. There are some places where blank spaces must not appear, specifically:

(1) within a line number
(2) within a reserved word
(3) within a numeric constant
(4) within a variable name

The following are some examples of incorrect use of blank spaces. The correct line appears in the column at the right.

(1) 1 00 PRINT"HELLO"
(2) 110 PR INT"HOW ARE YOU?"
(3) 120 LET A=1 00
(4) 130 LET CO ST=24.95

All reserved words in a program should be immediately preceded and followed by one of the following:

a blank space
an arithmetic operator (+ - * / ^)
the string operator (&)
a special character used in a particular statement format (<=>(),;:#)
end of line (ENTER key)

Examples:

```
>100 PRINT "HELLO"
>110 PRINT "HOW ARE YOU?"
>120 LET A=100
>130 LET COST=24.95
```

Line Numbers

Each program is comprised of a sequence of BASIC language program lines ordered by line number. The line number serves as a label for the program line. Each line in the program begins with a line number which must be an integer between 1 and 32767, inclusive. Leading zeroes may be used but are ignored by the computer. For example: 033 and 33 will be read as 33. You need not enter lines sequential order; they will be automatically placed that way by the computer.

When you run the program, the program lines are performed in ascending sequential order until:

(1) a branch instruction is performed (see "General Program Statements")
(2) an error occurs which causes the program to stop running (see "Error Messages")
(3) the user interrupts the running of the program with a BREAK command or by using the BREAK key (or CLEAR)
(4) a STOP statement or END statement is performed
(5) the statement with the largest line number is performed

If you enter a program line with a line number less than 1 or greater than 32767, the message BAD LINE NUMBER will be displayed and the line will not be entered into memory.

```
>NEW

>100 A=27.9
>110 B=31.8
>120 PRINT A;B
>RUN
  27.9  31.8

 ** DONE **

>0 A=2

 * BAD LINE NUMBER

>33000 C=4

 * BAD LINE NUMBER
```

Numeric Constants

Numeric constants must be either positive or negative real numbers. You may enter numeric constants with any number of digits. Values are maintained internally in seven radix-100 digits. This means that numbers will have 13 or 14 decimal digits depending on the value of the number.

Scientific Notation

Very large or very small numbers are easily handled using scientific notation. A number in scientific notation is expressed as a base number (mantissa) times ten raised to some power (exponent).

$$\text{Number} = \text{Mantissa} \times 10^{\text{Exponent}}$$

To enter a number using scientific notation:

First, type the mantissa (be sure to type a minus sign first if it's negative).

Type the letter "E" (must be an upper-case E).

Type the power of 10 (if it is negative, type the minus sign before you type the exponent).

The following are some examples of how numbers in scientific notation are entered.

Number	Entered as
3.264×10^{4}	3.264E4
-98.77×10^{21}	-98.77E21 or -9.877E22
5.691×10^{-5}	5.691E-5
-2.47×10^{-17}	-2.47E-17

Numeric constants are defined in the range of -9.99999999999999E127 to -1E-128, 0, and 1E-128 to 9.99999999999999E127.

Underflow--If an number is entered or computed whose value when rounded is greater than -1E-128 and less than 1E-128, an underflow occurs. When an underflow occurs, the computer replaces the value of the number with a zero and the program continues running. No warning or error is given.

Overflow--If a number is entered or computed whose value when rounded is greater than 9.99999999999999E127 or less than -9.99999999999999E127, an overflow occurs. When an overflow occurs, the constant is replaced by the computer's limit, a warning is given with the message NUMBER TOO BIG, and the program continues running. The computer's limit is -9.99999999999999E127 or 9.99999999999999E127 as appropriate. Note that "**" is printed if the exponent is greater than 99.

```
>PRINT 1.2
  1.2

>PRINT -3
 -3

>PRINT 0
  0
```

```
>PRINT 3.264E4
  32640

>PRINT -98.77E21
 -9.877E+22
```

```
>PRINT -9E-130
  0

>PRINT 9E-142
  0
```

```
>PRINT 97E136

 * WARNING:
   NUMBER TOO BIG
  9.99999E+**

>PRINT -108E144

 * WARNING:
   NUMBER TOO BIG
 -9.99999E+**
```

String Constants

A string constant is a string of characters (including letters, numbers, spaces, symbols, etc.) enclosed in quotes. Spaces within string constants are not ignored and are counted as characters in the string. All characters on the keyboard that can be displayed may be used in a string constant. A string constant is limited by the length of the input line (112 characters or four lines on the screen).

When a PRINT or DISPLAY statement is performed, the surrounding quote marks are not displayed. If you wish to have words or phrases within a string printed with surrounding quote marks, simply enter a pair of adjacent quote marks (double quotes) on either side of the particular word or phrase when you type it. Thus, three pairs of quotes are used in all.

```
>NEW

>100 PRINT "HI!"
>110 PRINT "THIS IS A STRING
 CONSTANT."
>120 PRINT "ALL CHARACTERS (+
 -*/ @,) MAY BE USED."
>RUN
 HI!
 THIS IS A STRING CONSTANT.
 ALL CHARACTERS (+-*/ @,) MAY
  BE USED.

 ** DONE **


>NEW

>100 PRINT "TO PRINT ""QUOTE
 MARKS"" YOU MUST USE DOUBLE
 QUOTES WITHIN A STRING."
>110 PRINT
>120 PRINT "TOM SAID, ""HI, MARY!"""
>RUN
 TO PRINT "QUOTE MARKS" YOU MUST USE DOUBLE QUOTES.

 TOM SAID, "HI, MARY!"

 ** DONE **
```

In BASIC all variables are given a name. Each variable name may be from one to fifteen characters in length but must begin with a letter, an at-sign (@), a left-bracket ([), a right-bracket (]), a back slash ( ), or an underline ( ). The only characters allowed in a variable name are letters, numbers, the at-sign (@), the underline (_), and the dollar sign ($).

The dollar sign must be the last character is a string variable name, and this is the only place in a variable name that it may be used. Variable names are restricted to fifteen characters, including the dollar sign for string variable names.

Array names follow the same rules as simple variable names. (See the section on Arrays for more information.) In a single program, the same name cannot be used both as a simple variable and as an array name, nor can two arrays with different dimensions have the same name. For example, Z and Z(3) cannot both be used as names in the same program, nor can X(3,4) and X(2,1,3). However, there is no relationship between a numeric variable name and a string variable name are the same except for the dollar sign (X and X$ may both be used in the same program).

Numeric Variable Names

Valid: X, A9, ALPHA, BASE_PAY, V(3), T(X,3), TABLE(X,XX7Y/2)
Invalid: X$, X/B, 3Y

String Variable Names

Valid: S$, YZ2$, NAME$, QS$(3,X)
Invalid: S$3, X9, 4Z$

If you enter a variable name with more than fifteen characters, the message BAD NAME is displayed and the line is not entered into memory. Reserved words are not allowed as variable names but may be used as part of a variable name. For example, LIST is not allowed as a variable name but LIST$ is accepted.

At any instant while a program is running, every variable has a single value. When a program begins running, the value associated with each numeric variable is set to zero and the value associated with each string variable is set to null (a string with a length of zero characters). When a program is running, values are assigned to variables when LET statements, READ statements, FOR TO STEP statements, or INPUT statements are performed. The length of the character string value associated with a string variable may vary from a length of zero to a limit of 255 characters while a program is running.

```
>110 ABCDEFGHIJKLMNOPQ=3
* BAD NAME
```

Numeric Expressions

Numeric expressions are constructed from numeric variables, numeric constants, and function references using arithmetic operators (+-*/^). All functions referenced in an expression must be either functions supplied in TI-99/2 BASIC (see sections on Built-In Functions) or defined by a DEF statement. The two kinds of arithmetic operators (prefix and infix) are discussed below.

The prefix arithmetic operators are plus (+) and minus (-) and are used to indicate the sign (positive or negative) of constants and variables. The plus sign indicates the number following the prefix operator (+) should be multiplied by +1, and the minus sign indicates the number following the prefix operator (-) should be multiplied by -1. Note that if no prefix operator is present, the number is treated as if the prefix operator were plus. Some examples of prefix operators with constants and variables are:

```
10  -6  +3
+A    -W
```

The infix arithmetic operators are used for calculations and include: addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). An infix operator must appear between each numeric constant and/or variable in a numeric expression. Note that multiplication cannot be implied by simply placing variables side by side or by using parentheses. You must use the multiplicaton operator (*).

Infix and prefix operators may be entered side by side within a numeric expression. The operators are evaluated in the normal way.

In evaluating numeric expressions, TI-99/2 BASIC uses the standard rules for mathematical hierarchy. These rules are outlined here.

1. All expressions within parentheses are evaluated first according to the hierarchical rules.
2. Exponentiation is performed next in order from left to right.
3. Prefix plus and minus are performed.
4. Multiplications and divisions are then completed.
5. Additions and subtractions are then completed.

Note that 0^0 is defined to be 1 as in ordinary mathematical usage.

In the evaluation of a numeric expression if an underflow occurs, the value is simply replaced by zero and the program continues running. If an overflow occurs in the evaluation of a numeric expression, the value is replaced by the computer's limit, a warning condition is indicated by the message "WARNING: NUMBER TOO BIG," and the program continues running.

When evaluation of a numeric expression results in division by zero, the value is replaced by the computer's limit with the same sign as the numerator, the message WARNING: NUMBER TOO BIG is displayed, and the program continues running. If the evaluation of the operation of exponentiation results in zero being raised to a negative power, the value is replaced by the positive value of the computer's limit, the message WARNING: NUMBER TOO BIG is displayed, and the program continues running. If the evaluation of the operation of exponentiation results in a negative number being raised to a non-integral power, the message BAD VALUE is displayed, and the program stops running.

```
>NEW

>100 A=6
>110 B=4
>120 C=20
>130 D=2
>140 PRINT A*B/2
>150 PRINT C-D*3+6
>RUN
  12
  20
 ** DONE **


>PRINT 3+-1
  2

>PRINT 2*-3
 -6

>PRINT 6/-3
 -2


>NEW
>100 A=2
>110 B=3
>120 C=4
>130 PRINT A*(B+2)
>140 PRINT B^A-4
>150 PRINT -C^A;(-C)^A
>160 PRINT 10-B*C/6
>RUN
  10
  5.
 -16   16
  8

 ** DONE **


>PRINT 0^0
  1
```

```
>NEW

>100 PRINT 1E-200
>110 PRINT 24+1E-139
>120 PRINT 1E171
>130 PRINT (1E60*1E76)/1E50
>RUN
  0
  24

 * WARNING:
   NUMBER TOO BIG IN 120
  9.99999E+**

 * WARNING
   NUMBER TOO BIG IN 130
  1.E+78

 ** DONE **


>NEW

>100 PRINT -22/0
>110 PRINT 0^-2
>120 PRINT (-3)^1.2
>RUN

 * WARNING:
   NUMBER TOO BIG IN 100
-9.99999E+**

* WARNING:
  NUMBER TOO BIG IN 110
 9.99999E+**

* BAD VALUE IN 120
```

Relational Expressions

Relational expressions are normally used in the IF THEN ELSE statement but may be used anywhere numeric expressions are allowed. When you use relational expressions within a numeric expression, a numeric value of -1 is given if the relation is true and a numeric value of 0 is given if the relation is false.

Relational operations are performed from left to right before string concatenation and after all arithmetic operations within the expression are completed. To perform string concatenation before relational operations and/or to perform relational operations before arithmetic operations, you must use parentheses. Valid relational operators are:

Equal to (=)	Not equal to (<>)
Less than (<)	Less than or equal to (<=)
Greater than (>)	Greater than or equal to (>=)

An explanation of how string comparisons are performed to give you a true or false result is discussed in the IF THEN ELSE explanation. Remember that the result you obtain from the evaluation of a relational operator is always a number. If you try to use the result as a string, you will get an error.

```
>NEW

>100 A=2<5
>110 B=3<=2
>120 PRINT A;B
>RUN
 -1   0

 ** DONE **


>NEW
>100 A$="HI"
>110 B$=" THERE!"
>120 PRINT (A$&B$)="HI!"
>RUN
   0

 ** DONE **

>120 PRINT (A$&B$)>"HI"
>RUN
 -1

 ** DONE **

>120 PRINT (A$>B$)*4
>RUN
 -4

 ** DONE **


>NEW

>100 A=2<4*3
>110 B=A>0
>120 PRINT A;B
>RUN
 -1   0

 ** DONE **
```

String Expressions

String expressions are constructed from string variables, string constants, and function references using the operation for concatenation (&). The operation of concatenation allows you to combine strings together. All functions referenced in a string expression must be either functions supplied in TI-99/2 BASIC (see Built-In String Functions) or defined by a DEF statement and must have a string value. If evaluation of a string expression results in a value which exceeds the maximum string length of 255 characters, the string is truncated on the right, and the program continues running. No warning is given.

Note that all characters included in a string expression are always displayed on the screen exactly as you enter them.

```
>NEW

>100 A$="HI"
>110 B$="HELLO THERE!"
>120 C$="HOW ARE YOU?"
>130 MSG$=A$&SEG$(B$,6,7)
>140 PRINT MSG$&" "&C$
>RUN
 HI THERE!  HOW ARE YOU?

 ** DONE **
```

INDEX