TEXAS INSTRUMENTS

PROGRAMMABLE

TI-74

PROGRAMMING REFERENCE GUIDE

TEXAS INSTRUMENTS

TI-74

PROGRAMMING REFERENCE GUIDE

Manuals developed by: The staff of Texas Instruments Instructional Communications

Jacquelyn F. Quiram
Kenneth E. Heichelheim
Mike Keller
Brenda Cornitius
Chris M. Alley

With contributions by:

Bruno Didier
Robert A. Pollan
Stephen L. Reid
Floyd R. Gerwig
Robert E. Whitsitt, II

Table of Contents

This guidebook contains information about the BASIC programming language of the Texas Instruments TI-74 BASICALC™ calculator. You can find operating instructions in the TI-74 User's Guide.

Chapter 1: Overview of BASIC

Use this chapter if you know the type of operation you want to perform but do not know the keyword. After you locate the keywords that apply to that type of operation, refer to Chapter 2 for a detailed description of each keyword and an example of how to use it.

Table of Contents

General Programming Information

This section briefly describes some of the conventions and features of TI-74 BASIC.

Line Numbers

Each program line must begin with a line number. The line number must be an integer in the range of 1 to 32766. The following keywords enable you to generate new line numbers and to change existing ones.

NUM	Automatically generates program line numbers as you enter a program.
REN	Renumbers the lines of a program.

Duplicating a Program Line

You can duplicate a program line by recalling an existing line and typing a new line number in place of the old number. This creates a new line without affecting the original line.

Multiple-Statement Lines

In general, you can include more than one statement in a program line. The statements must be separated by a colon.

```
100 X = 3:PRINT X:PAUSE
```

The following restrictions apply to multiple-statement lines.

- You cannot use an IMAGE or DATA statement in a multiple-statement line.
- If you use the SUB statement in a multiple-statement line, it must be the first statement in the line.
- If you use a remark in a multiple-statement line, it must be the last statement in the line.
- If you use a DIM statement in a multiple-statement line, it must be the last statement in the line. (You can use only one DIM statement in a line.)

Remarks

You can document your programs by using remark statements. A remark is a reminder for the programmer and is ignored by the computer during program execution.

REM	Signifies that the rest of a program line is a remark.
!	Can be substituted for the REM keyword.

Memory Management

TI-74 BASIC contains two keywords that help you manage memory space.

ADDMEM	Appends the memory in a RAM cartridge to the built-in memory of the TI-74. (Because this is a built-in subprogram, use CALL ADDMEM.)
FRE	Indicates the total amount of memory available for program and data storage, or reports the amount of memory used by the current program.

Assigning Values to Variables

You can assign values to variables from data either contained within the program or entered from the keyboard during program execution.

Variable Names

A valid variable name must follow the rules listed below.

- A variable name can have as many as 15 characters.
- The first character must be a letter, underline, or @ symbol. The remaining characters can be letters, numbers, underlines, or @ symbols.
- A string variable name must end with a $ symbol. (You cannot use the $ symbol as part of a numeric variable name.)

Data Contained Within a Program

The following statements enable you to assign values to variables by using data contained within a program.

READ	Assigns values to variables by sequentially reading values from DATA lists.
DATA	Provides a list of values to be assigned to variables by READ statements.
RESTORE	Determines which DATA statement will be read next.

Data Entered From the Keyboard

The following statements suspend program execution and enable you to enter data from the keyboard.

INPUT	Waits for you to enter one or more values. INPUT can display a prompt.
LINPUT	Waits for you to enter a string value. LINPUT accepts your entry exactly as typed, including commas and quotes. LINPUT can display a prompt.
ACCEPT	Waits for you to enter a value. ACCEPT has several options that are not available with INPUT or LINPUT.

Displaying Information

The keywords in this section enable you place information in the display.

Unformatted Display

The following keywords enable you to display numeric and string values.

PRINT	Displays a constant, the value of a variable, or the result of an expression. (You normally use a PAUSE following the PRINT statement.)
DISPLAY	Similar to PRINT, except that DISPLAY has several options that make it more versatile than PRINT. (You normally use a PAUSE following the DISPLAY statement.)
PAUSE	Suspends program execution for a specified period of time to allow you to view the information in the display. Without PAUSE, results are displayed too quickly to be viewed.

INPUT and LINPUT also allow you to display a prompt or message related to the desired input.

Formatted Display

You can use several options with PRINT and DISPLAY to control the way information is displayed.

USING	Defines a format string for displayed information or references a format string contained in a separate IMAGE statement.
IMAGE	Allows a format string to be placed on a separate line so that it can be referenced by more than one PRINT USING or DISPLAY USING statement.
TAB	Positions information at a specified column in the display.

Numeric Operations

TI-74 BASIC includes a wide variety of mathematical operators and functions.

Arithmetic Operators

You can use the following operators in a numeric expression.

A+B	Adds A and B.
A−B	Subtracts B from A.
A * B	Multiplies A and B.
A / B	Divides A by B.
A ^ B	Raises A to the power of B.

+ and − can also be used as unary operators to indicate positive or negative values, such as +4 or −6.

The TI-74 also has relational and logical operators that let you compare two numeric expressions. For information on these operators, refer to page 1-12.

Hierarchy of Operators

The following list shows the order in which arithmetic operations are performed.

unary +, unary −
^
*, /
+, −

An expression enclosed in parentheses is given priority over operations outside the parentheses.

Trig Functions

The following keywords select an angle setting for trigonometric calculations. All angles are interpreted according to the current setting.

DEG	Selects the degree setting.
RAD	Selects the radian setting.
GRAD	Selects the grad setting.

Always begin trig calculations by selecting the appropriate angle setting. You can then use the following trig functions.

ACOS	Computes the arccosine of a number.
ASIN	Computes the arcsine of a number.
ATN	Computes the arctangent of a number.
COS	Computes the cosine of an angle.
SIN	Computes the sine of an angle.
TAN	Computes the tangent of an angle.

Hyperbolic Functions

TI-74 BASIC includes the following hyperbolic functions.

ACOSH	Computes a hyperbolic arccosine.
ASINH	Computes a hyperbolic arcsine.
ATANH	Computes a hyperbolic arctangent.
COSH	Computes a hyperbolic cosine.
SINH	Computes a hyperbolic sine.
TANH	Computes a hyperbolic tangent.

Numeric Operations (Continued)

Other Numeric Functions

Other numeric functions available in TI-74 BASIC are listed below.

ABS	Computes absolute value.
EXP	Computes e raised to a power.
INT	Returns the integer portion of a number.
LN	Computes natural logarithm.
LOG	Computes common logarithm.
PI	Returns the value of pi as 3.141592654.
RANDOMIZE	Ensures that a random number is unpredictable by setting a random starting value (seed) for the random number generator.
RND	Returns a random number.
SGN	Tests a number to see if it is positive, negative, or zero.
SQR	Computes square root.

String Operations

The computer's string operations enable you to include prompts and messages in your programs and to write programs that process string information.

String Constants

When you type a string constant, enclose it in quotation marks. The quotation marks identify the string, but they are not considered part of the string. If you want to include a quotation mark within a string, type a pair of consecutive quotes.

String Typed	String Displayed
"Quote Test"	Quote Test
"Quote""Test"	Quote"Test
"""Quote""""Test"""	"Quote""Test"

To enter a null (empty) string, type the open and close quotation marks with no characters between them.

Concatenating Strings

The concatenation operator (&) combines two strings into a single string. The resulting string cannot have more than 255 characters.

The following program segment shows an example of string concatenation. (Notice that the first character of STRING2$ is a blank space, so that the concatenated string is properly spaced.)

```
100 STRING1$ = "THIS IS A"
110 STRING2$ = " CONCATENATED STRING"
120 STRING3$ = STRING1$ & STRING2$
130 PRINT STRING3$:PAUSE
```

The program segment displays:

```
THIS IS A CONCATENATED STRING
```

The TI-74 also has relational and logical operators that let you compare two string expressions. For information on these operators, refer to page 1-12.

String Functions

The following string functions operate on a string expression and return a numeric value.

ASC	Converts the first character of a string to its corresponding ASCII character code.
NUMERIC	Tests a string expression to see if it is a valid representation of a numeric value.
VAL	Returns the numeric value of a string expression.
LEN	Calculates the length of a string.
POS	Searches a string for the first occurrence of a substring.

The following string functions operate on a numeric expression and return a string value.

CHR$	Converts an ASCII character code to its corresponding display character.
STR$	Converts a number into a string.

The following string functions operate on a string expression and return a string value.

RPT$	Creates a string by repeating a starting string a specified number of times.
SEG$	Returns a substring of a string.

Arrays

TI-74 BASIC allows one-, two-, and three-dimensional arrays.

Defining the Size of an Array

An array requires space in memory. You can define an array with the DIM statement, or you can accept the default provided by the TI-74.

DIM	Defines the number of dimensions and the number of elements in each dimension of an array.

If you refer to an array that has not been defined by a DIM statement, the TI-74 automatically sets an array size of 11 elements (numbered 0 through 10) for each dimension in the array.

For example, if you use the following statement without first dimensioning the array SOMENAME, the array defaults to two dimensions with each dimension having space for eleven elements.

SOMENAME(0,0) = 123

If your program references an array element outside the range established for the array, the TI-74 returns an error message. If you enter a non-integer as a subscript, the TI-74 rounds the number. If you enter a negative integer as a subscript, the TI-74 returns an error message.

Relational and Logical Operators

A relational expression compares two numeric or string values and then checks to see if a specified relationship is true or false. A logical expression connects two relational expressions.

Relational Operators

The following table shows the six relational operators. In each example, the operator compares the values of A and B, which represent either numeric or string expressions.

A < B	True if A is less than B.
A <= B	True if A is less than or equal to B.
A > B	True if A is greater than B.
A >= B	True if A is greater than or equal to B.
A = B	True if A is equal to B.
A <> B	True if A is not equal to B.

Note: In most programs, it is important only that a condition is true or false. However, some programs can take advantage of the fact that a relational expression evaluates to −1 if the condition is true or 0 if the condition is false.

String Comparisons

When a relational expression compares two string values, the TI-74 takes one character at a time from each string and compares their ASCII codes. Leading and trailing blanks are included in the comparison. (For a list of ASCII codes, refer to Appendix B.)

- If the ASCII codes differ, the string with the lower code is less than the string with the higher code.
- If all the ASCII codes are the same and both strings are the same length, the strings are equal.
- If one of the strings is longer, the comparison is performed for as many characters as there are in the shorter string. If all the ASCII codes are the same, the longer string is considered greater.
- The null string ("") is less than every other string.

Logical Operators

The following table shows examples using the four logical operators.

A AND B	True only if both A and B are true.
A OR B	True if either A or B is true, or if both A and B are true.
A XOR B	True if either A or B is true, but false if both A and B are true.
NOT A	True only if A is false.

The logical operators can also be used to manipulate individual bits of a numeric value. For more information, refer to Appendix C.

Control Statements

Control statements enable you to alter the order in which program statements are executed. Control statements can result in unconditional branching, conditional branching, or looping.

Unconditional Branching

An unconditional branching statement always tranfers control to a specified line number.

GOTO	Transfers control to one and only one specified line number.

Conditional Branching

A conditional branching statement enables a program to select one of several alternative paths, depending on certain conditions within the program.

IF/THEN/ELSE	Uses relational and logical operators to test a condition in the program and determine which statements to execute.
ON GOTO	Transfers control to one of several line numbers, depending on the value of a numeric expression.
ON GOSUB	Transfers control to one of several subroutines, depending on the value of a numeric expression.

Looping

A FOR/NEXT loop repeats the statements in the loop a specified number of times. The following keywords enable you to set up a FOR/NEXT loop.

FOR/TO/STEP	Marks the beginning of a FOR/NEXT loop.
NEXT	Marks the end of a FOR/NEXT loop. When the NEXT statement is executed, control returns to the statement immediately following the FOR/TO/STEP statement.

Subroutines and Subprograms

As your programs become more complicated, you may need to use the same group of lines at several different places. Instead of duplicating the lines at each place, it is more convenient to enter them as a subroutine or a subprogram. A program can access a subroutine or a subprogram from any place within the program.

Using a Subroutine

The following keywords enable you to use subroutines.

GOSUB	Transfers control from the main program to a subroutine that begins at a specified line number.
ON GOSUB	Transfers control to one of several subroutines, based on the value of a numeric expression.
RETURN	Marks the end of the subroutine, and transfers control back to the statement that follows the GOSUB or ON GOSUB statement.

It is common practice to place subroutines after the main sequence of a program. However, it is invalid to place subroutines after the END statement.

Using a Subprogram

The following keywords enable you to define and access a subprogram.

CALL	Transfers control from the main program to a specified subprogram name.
SUB	Labels the beginning of a subprogram.
SUBEND	Marks the end of a subprogram.
SUBEXIT	Terminates the execution of a subprogram. This statement is used when you want to exit the subprogram before the SUBEND statement.

A subprogram must be placed after the last statement in the main program.

Built-In Subprograms

The TI-74 has six built-in subprograms—ADDMEM, ERR, GET, IO, KEY, and PUT. To access these subprograms, remember to use the CALL statement. For example, use CALL ADDMEM to access the ADDMEM subprogram.

Error Handling

The TI-74 has several statements that help you locate errors in the program. Some statements help you debug the program, and other statements enable you to trap any errors that may occur during program execution.

Debugging a Program

When debugging a program, you can use the following keywords to interrupt program execution so you can test the value of variables in the program.

BREAK	Sets breakpoints within a program. **Note:** You can also press the **[BREAK]** key to interrupt program execution.
CON	Continues program execution following a breakpoint.
ON BREAK	Enables you to select the action taken when a breakpoint occurs.
UNBREAK	Removes the breakpoints set with the BREAK statement.

Handling Errors and Warnings

By processing errors and warnings, a program can often correct problems before they interfere with program execution.

ON ERROR	Enables you to determine the action taken when an error occurs during program execution.
ON WARNING	Enables you to determine the action taken when a warning occurs during program execution.

Storing and Retrieving Programs

You may want to save a copy of your program for future use by storing it on an external device such as a cassette player/recorder or a RAM cartridge. After it is stored, you can easily retrieve the program at any time and reload it into the TI-74. (For detailed information on using cassette tapes, refer to Chapter 3 in this book.)

Storing a Program

The following keywords enable you to store a copy of the program that is currently in the TI-74's memory.

FORMAT	Initializes the storage medium on an external device. (Cassette tapes do not need to be formatted.) **Caution:** If a medium that already contains stored programs is reformatted, those programs are erased.
SAVE	Copies the program to an external storage device, such as a cassette recorder.
VERIFY	Checks the copy stored on the external device to make sure that it was copied correctly.
PUT	Copies the program to a RAM cartridge. (Because this is a built-in subprogram, use CALL PUT.)

Retrieving a Program

The following keywords enable you to retrieve a copy of a program stored on an external device.

Caution: When you retrieve a program from an external device, any program currently stored in the TI-74's memory is erased. If you want to save the current program, be sure to store it before retrieving the new program.

OLD	Retrieves a program from an external device. such as a cassette player.
GET	Retrieves a program from a RAM cartridge. (Because this is a built-in subprogram, use CALL GET.)

Storing and Retrieving Data Files

When a program processes a large amount of data, it is usually more convenient to store the data on an external device such as a cassette player/recorder.

Naming a Data File

The valid name for a file opened on an external device depends on the design of the peripheral. For cassette files, a file name can have from 1 to 18 characters. The name must start with a letter but can have any character except comma or period in the rest of the name. Cassette file names can include a .NM extension to omit messages.

Using Data Files

The following keywords enable you to store and retrieve data records. You can also delete data files.

FORMAT	Initializes the storage medium on an external device. (Cassette tapes do not need to be formatted.) **Caution:** If you reformat a medium that already contains stored programs, those programs are erased.
OPEN #	Opens a communication link between the TI-74 and the external device.
CLOSE #	Closes the communication link between the TI-74 and the external device.
PRINT #	Stores a record on an open data file.
INPUT #	Retrieves a record from an open data file.
EOF	Tests the data file to see if there are any remaining records in the file.
RESTORE	Selects the next data record to be input. (You cannot use RESTORE with a cassette player/recorder.)
DELETE	Deletes a specified data file from an external device.

Sending Information to a Printer

By attaching an optional printer to the TI-74, you can print a listing of the program currently in memory. You can also include statements in your programs to send information to the printer.

Listing a Program

You can list a program by using a single keyword. You do not have to OPEN and CLOSE the printer for a listing.

LIST	Prints a line-by-line listing of the program. LIST must be used as a command; it cannot be used in a program.

Using a Printer From Within a Program

The following keywords can be used as statements within a program.

OPEN #	Opens a communication link between the TI-74 and the printer.
CLOSE #	Closes the communication link between the TI-74 and the printer.
PRINT #	Sends a number or string to the printer.
IO	Performs additional control operations that are not built into TI-74 BASIC. (Because this is a built-in subprogram, use CALL IO.) The operations possible with the IO subprogram depend on the design of your printer.

Refer to "Controlling the Printer From BASIC" in Chapter 3 of this manual for information about using the PC-324 thermal printer.

Chapter 2: Reference Section

This chapter describes each command, statement, and function of TI-74 BASIC. The keywords are presented in alphabetical order.

Table of Contents

Conventions in this Chapter

All descriptions of the BASIC keywords follow the same order of presentation. The order and the conventions used in the descriptions are explained below.

Order

The purpose of the keyword is stated first.

The Format section gives the complete syntax of the keyword.

The Description section explains the keyword's use or function and includes the options that the keyword can use.

The Example section gives examples of the keyword's use, where appropriate.

The Cross Reference section refers to similar and complementary keywords, where appropriate.

Format Conventions

The Format sections use the following conventions.

- KEYWORDS are capitalized.
- *Variables* are in italics.
- Optional items are enclosed in brackets ([ ]).
- All parentheses are required. Parentheses included with an optional item must be included when the optional item is used.
- Items that may be repeated are indicated by ellipses (. . .).

ABS

The ABS function computes the absolute value of a numeric expression. The absolute value is always a positive number or zero.

Format

ABS(*numeric-expression*)

Description

The ABS function operates on the numeric expression as described below:

- If *numeric-expression* is positive or zero, ABS returns the value.
- If *numeric-expression* is negative, ABS returns the negative of the value.

Examples

```
140 PRINT ABS(42.3) : PAUSE
```
Prints 42.3.

```
370 V=ABS(-6.124)
```
Sets V equal to 6.124.

Cross Reference

SGN

ACCEPT

The ACCEPT statement suspends program execution and enables you to enter data from the keyboard. This statement can accept data at any column in the display, erase all or part of the display, limit the number and type of characters accepted, and provide a default value for the input.

Format

ACCEPT [[AT(*column*)] [SIZE(*numeric-expression*)]
[ERASE ALL] [VALIDATE(*data-type*,...)]
[NULL(*expression*)],]*variable*

Description

The general form of the ACCEPT statement

ACCEPT *variable*

assigns the data entered from the keyboard to *variable*. The variable can be either numeric or string, depending on the type of data to be entered. You can enter a maximum of 80 characters, and any trailing spaces are ignored.

The display is cleared from the current cursor position to the end of the 80-column line. Input begins in column 1 unless a pending input/output statement positions the cursor elsewhere, in which case input is accepted at the cursor location.

Note: To enter a string that contains a comma, a quotation mark, or leading or trailing spaces, you must enclose the string in quotation marks. A quotation mark within a string is represented by two adjacent quotation marks.

When an ACCEPT statement is waiting for data, **[CLR]** clears only the input field; **[CTL][↑]** (home) and **[CTL][←]** (back tab) move the cursor to the beginning of the input field; and **[CTL][→]** has no effect.

Options

You can use one or more of the following options, in any order, with the ACCEPT statement. Precede each option by a space (unless it is preceded by the close parenthesis ")" of a previous option), and be sure to place a comma after the last option before *variable*.

- AT(*column*)—positions the cursor at the column specified by the rounded value of *column*. The input field extends from the cursor position to the end of the 80-column line.

  Valid *column* values range from 1 through 31. If the rounded value of *column* is outside this range, the error message `Bad value` is displayed.

- SIZE(*numeric-expression*)—limits the maximum number of characters that can be entered. You can enter up to the absolute value of *numeric-expression* characters.

  If the value of *numeric-expression* is positive, the input field is cleared before input is accepted.

  If the value of *numeric-expression* is negative, the input field is not cleared. This allows you to use a previous DISPLAY or PRINT statement to place a default value into the input field.

  The cursor is left in the first position following the input field for subsequent input/output statements.

  Note that the SIZE option specifies only the maximum number of characters that can be input. Regardless of the specified size, the input field cannot extend beyond the end of the 80-column line.

- ERASE ALL—clears the entire 80-column display line before accepting input.

Options (Continued)

- VALIDATE(*data-type*)—allows you to enter only the characters specified by *data-type*. If more than one *data-type* is specified, a character from any of the types is acceptable.

  Data-type can be one of the types shown below.

Type	Valid Input
ALPHA	All alphabetic characters
UALPHA	Only uppercase alphabetic characters
DIGIT	All digits (0—9)
NUMERIC	All digits (0—9), the decimal point (.), the plus sign (+), the minus sign (−), and the uppercase letter E
ALPHANUM	All alphabetic characters and digits
UALPHANUM	Only uppercase alphabetic characters and digits

  Note: When using the UALPHA and UALPHANUM data-types, any lowercase alphabetic characters entered are automatically accepted as uppercase characters.

  Data-type can also be a string expression, in which case any character or combination of characters in the string is acceptable as input.

- NULL(*expression*)—provides a default value specified by *expression*. If you press **[ENTER]** with a blank (or null) input field, the default value is automatically assigned to *variable*. Note that the default value is not affected by the VALIDATE option.

Additional Entry Methods

There are two additional methods of entering data during the execution of an ACCEPT statement. You can:

- Use the **[FN]** key to input keywords and user-assigned strings.
- Enter a numeric expression if *variable* is numeric. The expression is evaluated and the result is assigned to *variable*.

Examples

```
100 ACCEPT AT(5)ERASE ALL,T
```
Clears the display and accepts data starting in column 5 through the end of the line. The data is assigned to the variable T.

```
320 ACCEPT VALIDATE("YN") SIZE(1),A$
```
Accepts a one character field consisting of either Y or N. The character is assigned to the variable A$.

```
430 ACCEPT AT(3)SIZE(-5) VALIDATE(DIGIT,"+-"),X
```
Accepts up to 5 characters, beginning in column 3, for the variable X. The input characters must consist of digits or the characters + or −. The input field is not erased because the SIZE specification is negative.

```
570 ACCEPT NULL(PI),C
```
Accepts data for the variable C. If no data has been entered when you press **[ENTER]**, the value of PI is stored in C.

```
210 DISPLAY "ADDRESS:";:ACCEPT AT(10),ADDR$
```
Shows ADDRESS: and positions the cursor after the prompt. Accepts data for the variable ADDR$. Notice that a pause is not needed with DISPLAY because the semicolon creates a pending print condition.

Cross Reference

INPUT, LINPUT

ACOS

The ACOS function computes the arccosine of a numeric expression.

Format ACOS(*numeric-expression*)

Description The ACOS function returns the angle whose cosine is *numeric-expression*. This angle is interpreted as radians, degrees, or grads according to the current setting of the angle units (RAD, DEG, or GRAD).

The value of *numeric-expression* must be in the range of −1 through 1. Otherwise, the error message `Bad argument` is displayed.

The range of values returned by the ACOS function for the three angle settings is shown below.

Angle Setting	Range of Values
DEG	$0 \leq ACOS(X) \leq 180$
RAD	$0 \leq ACOS(X) \leq PI$
GRAD	$0 \leq ACOS(X) \leq 200$

Examples

```
100 DEG:PRINT ACOS(1):PAUSE
```
Prints 0. (The cosine of 0 degrees is 1.)

```
220 RAD:T=ACOS(.75)
```
Sets T equal to .7227342478.

Cross Reference ASIN, ATN, COS, DEG, GRAD, RAD, SIN, TAN

ACOSH

The ACOSH function computes the hyperbolic arccosine of a numeric expression.

Format ACOSH(*numeric-expression*)

Description The ACOSH (hyperbolic arccosine) function returns the number whose hyperbolic cosine is *numeric-expression*. The definition of hyperbolic arccosine is:

ACOSH(X) = LN(X + SQR(X*X − 1))

Examples

```
100 PRINT ACOSH(1):PAUSE
```
Prints 0 (the hyperbolic arccosine of 1).

```
230 A=ACOSH(4/3)
```
Sets A equal to .7953654612 (the hyperbolic arccosine of 4/3).

Cross Reference ASINH, ATANH, COSH, SINH, TANH

ADDMEM Subprogram

The ADDMEM subprogram appends the Random Access Memory (RAM) contained in an installed 8K Constant Memory™ cartridge to the available resident memory.

Format CALL ADDMEM

Description The ADDMEM subprogram expands resident memory to include the memory in a RAM cartridge. The TI-74 then treats the combined memory as continuous resident memory.

The ADDMEM subprogram can be used only as a command; it cannot be used in a program.

The memory capacity resulting from adding cartridge memory to resident memory is 16K bytes.

The error message `E31 No RAM` is displayed if no 8K Constant Memory cartridge is installed when CALL ADDMEM is executed.

When the memory in an 8K Constant Memory cartridge is appended to resident memory, the system retains stored BASIC program lines.

Before you execute an ADDMEM subprogram, the memory in an 8K Constant Memory cartridge is identified by the TI-74 as a separate memory area. The contents of resident memory can be stored in the cartridge by a PUT command. After an ADDMEM subprogram is executed, however, the TI-74 considers the cartridge to be part of resident RAM, and the cartridge memory is no longer available as a separate memory area.

The memory in an 8K Constant Memory cartridge remains appended to the resident memory until a NEW ALL command is executed, the computer is turned on without the cartridge installed, the batteries are removed, or the system is initialized.

Cross Reference GET, PUT

ASC

The ASC function converts the first character in a string to its equivalent ASCII character code.

Format ASC(*string-expression*)

Description The ASC function returns the ASCII code of the first character in *string-expression*. You can use any string except the null string, in which case the error message `Bad argument` is displayed.

ASC is the inverse of the CHR$ function. A list of the ASCII codes is given in Appendix B.

Examples

```
100 X = ASC("A")
```
Sets X equal to 65 (the ASCII code for the letter A).

```
130 DISPLAY ASC("HELLO"):PAUSE
```
Displays 72 (the ASCII code for the letter H).

```
790 DISPLAY ASC(A$):PAUSE
```
Displays the ASCII code for the first character in A$.

Cross Reference CHR$

ASIN

The ASIN function computes the arcsine of a numeric expression.

Format ASIN(*numeric-expression*)

Description The ASIN function returns the angle whose sine is *numeric-expression*. This angle is interpreted as radians, degrees, or grads according to the current setting of the angle units (RAD, DEG, or GRAD).

The value of *numeric-expression* must be in the range of −1 through 1. Otherwise, the error message `Bad argument` is displayed.

The range of values returned by the ASIN function for the three angle settings is shown below.

Angle Setting	Range of Values
DEG	$-90 \leq \text{ASIN}(X) \leq 90$
RAD	$-\text{PI}/2 \leq \text{ASIN}(X) \leq \text{PI}/2$
GRAD	$-100 \leq \text{ASIN}(X) \leq 100$

Examples

```
140 DEG:PRINT ASIN(1):PAUSE
```
Prints 90. (The sine of 90 degrees is 1.)

```
240 RAD:B=ASIN(.9)
```
Sets B equal to 1.119769515.

Cross Reference ACOS, ATN, COS, DEG, GRAD, RAD, SIN, TAN

ASINH

The ASINH function computes the hyperbolic arcsine of a numeric expression.

Format ASINH(*numeric-expression*)

Description The ASINH (hyperbolic arcsine) function calculates the number whose hyperbolic sine is *numeric-expression*. The definition of hyperbolic arcsine is shown below.

ASINH(X) = LN(X + SQR(X*X + 1))

Examples

```
100 PRINT ASINH(0):PAUSE
```
Prints 0.

```
230 A=ASINH(4/3)
```
Sets A equal to 1.098612289 (the hyperbolic arcsine of 4/3).

Cross Reference ACOSH, ATANH, COSH, SINH, TANH

ATANH

The ATANH function computes the hyperbolic arctangent of a numeric expression.

Format

ATANH(*numeric-expression*)

Description

The ATANH (hyperbolic arctangent) function calculates the number whose hyperbolic tangent is *numeric-expression*. The definition of hyperbolic arctangent is shown below.

ATANH(X) = .5*LN((1 + X)/(1 – X))

Examples

```
100 PRINT ATANH(0):PAUSE
```
Prints 0.

```
230 A=ATANH(4/7)
```
Sets A equal to .6496414921 (the hyperbolic arctangent of 4/7).

Cross Reference

ACOSH, ASINH, COSH, SINH, TANH

ATN

The ATN function computes the arctangent of a numeric expression.

Format

ATN(*numeric-expression*)

Description

The ATN function returns the angle whose tangent is *numeric-expression*. This angle is interpreted as radians, degrees, or grads according to the current setting of the angle units (RAD, DEG, or GRAD).

The range of values returned by the ATN function for the three angle settings is shown below.

Angle Setting	Range of Values
DEG	$-90 \leq ATN(X) \leq 90$
RAD	$-PI/2 \leq ATN(X) \leq PI/2$
GRAD	$-100 \leq ATN(X) \leq 100$

Examples

```
130 DEG:PRINT ATN(1):PAUSE
```
Prints 45. (The tangent of 45 degrees is 1.)

```
810 RAD:Q=ATN(2.5)
```
Sets Q equal to 1.19028995.

Cross Reference

ACOS, ASIN, COS, DEG, GRAD, RAD, SIN, TAN

BREAK

The BREAK statement suspends program execution at specified points, called breakpoints, in a program.

Formats

BREAK
BREAK *line-number-list*

Description

The two formats for the BREAK statement are described below.

- BREAK—causes an immediate breakpoint when the statement is executed.
- BREAK *line-number-list*—sets a breakpoint(s) immediately before the specified line(s). If more than one line-number is specified, the lines must be separated by a comma.

When a breakpoint occurs, the message BREAK is displayed. A breakpoint remains in the program until you edit or delete the line or use the UNBREAK statement.

BREAK is useful in debugging a program. When program execution halts at a breakpoint, you can print variables and perform calculations to determine why a program is not executing correctly. You can use the CONTINUE command to resume program execution.

The BREAK statement is often used as a command, in which case you must include a line number. The **[BREAK]** key also causes the program to halt as if a BREAK statement had been executed.

Examples

```
150 BREAK
```
Causes a breakpoint when the BREAK statement is executed.

```
100 BREAK 120,130
```
Causes breakpoints before execution of lines 120 and 130.

```
BREAK 10,400,130
```
Causes breakpoints before execution of lines 10, 400, and 130.

Cross Reference

CONTINUE, ON BREAK, UNBREAK

CALL

The CALL statement transfers control from the main program to a specified subprogram. After the subprogram is executed, control returns to the first statement following the CALL statement.

Formats

CALL *subprogram-name*
CALL *subprogram-name* (*argument-list*)

Description

The two formats for the CALL statement are described below.

- CALL *subprogram-name*—transfers program control to the first subprogram found with the given *subprogram-name*. The *subprogram-name* must be the name of an existing subprogram. Otherwise, the error message `E13 Not found` is displayed.
- CALL *subprogram-name* (*argument-list*)—uses the *argument-list* to pass data to and from a subprogram. The list may consist of one or more arguments separated by commas. The number and types of arguments in *argument-list* must match the parameters in the *parameter-list* of the subprogram. Otherwise, the error message `E23 Bad argument` or `E1 Syntax` is displayed.

Subprogram Priority

The order in which the computer searches for a specified subprogram is as follows.

1. Built-in subprograms such as ADDMEM, ERR, IO, and KEY
2. BASIC subprograms defined with the SUB statement
3. Subprograms located in software cartridges

Each built-in subprogram is discussed under its own entry in this chapter. BASIC subprograms are discussed in Chapter 1 and in this chapter under SUB.

Example

```
100 CALL KEY(K,S)
```
Calls the built-in subprogram KEY.

Cross Reference

ADDMEM, ERR, GET, IO, KEY, PUT, SUB, SUBEND, SUBEXIT

CHR$

The CHR$ function converts a specified ASCII character code to its corresponding display character.

Format

CHR$(*numeric-expression*)

Description

The CHR$ function returns the character that corresponds to the ASCII code specified by the rounded value of *numeric-expression*. The value of *numeric-expression* must be in the range of 0 through 32767. Otherwise, the error message `Bad argument` is displayed.

Because valid ASCII character codes only range from 0 through 255, any value of *numeric-expression* from 256 through 32767 is automatically reduced by 256 until the value represents a valid ASCII code.

The CHR$ function is commonly used in PRINT and DISPLAY statements to display the extended character set, such as the special graphics characters, which is not directly accessible from the keyboard. CHR$ is also used to send control codes to a peripheral device, such as a printer.

CHR$ is the inverse of the ASC function. A list of the ASCII codes is given in Appendix B.

Examples

```
440 PRINT CHR$(72):PAUSE
```
Prints H.

```
640 X$=CHR$(33)
```
Sets X$ equal to !.

Cross Reference

ASC

CLOSE

The CLOSE statement closes a specified file.

Formats

CLOSE *#file-number*
CLOSE *#file-number*,DELETE

Description

The CLOSE statement terminates the association between a file and its current file number.

The two formats for the CLOSE statement are described below.

- CLOSE *#file-number*—closes the file that was previously opened as *file-number*. If you attempt to close a file that is not open, the message `File error` is displayed.
- CLOSE *#file-number*,DELETE—closes and deletes the file that was previously opened as *file-number*. This format applies only to peripherals that allow you to delete files.

After a file is closed, it cannot be accessed by the program unless it is reopened. After a file is closed, *file-number* can be assigned to another file or device.

Additional Methods of Closing a File

To protect the data in your files, open files are also closed when you do any of the following.

- Edit the program or subprogram.
- Enter a NEW, OLD, RENUMBER, RUN, SAVE, or VERIFY command.
- List the program to a peripheral device.
- Call the ADDMEM subprogram.
- Turn the system off or press the **[RESET]** key.
- Press the **[MODE]** key to switch to CALC mode.

Normal program termination also closes all open files.

Example

```
790 CLOSE #6
```
Closes the file that was opened as file #6.

Cross Reference

DELETE, OPEN

CONTINUE

The CONTINUE (or CON) command resumes program execution following a breakpoint.

Formats

CONTINUE
CONTINUE *line-number*

Description

The two formats for the CONTINUE command are described below. The abbreviation CON may be used instead of CONTINUE.

- CON—resumes execution beginning at the program line that immediately follows the breakpoint.
- CON *line-number*—resumes execution beginning at the specified *line-number*.

  If the breakpoint occurred in the main program, *line-number* must refer to a line number in the main program. If the breakpoint occurred in a subprogram, *line-number* must refer to a line number in that subprogram. Entering an improper *line-number* produces unpredictable results.

Exceptions

If you do any of the following after the breakpoint occurs, the CON command will not resume program execution.

- Edit the program or subprogram.
- Enter a NEW, OLD, RENUMBER, RUN, SAVE, or VERIFY command.
- List the program to a peripheral device.
- Call the ADDMEM subprogram.
- Turn the system off or press the **[RESET]** key.
- Press the **[MODE]** key to switch to CALC mode.

Cross Reference

BREAK

COS

The COS function computes the trigonometric cosine of a numeric expression.

Format

COS(*numeric-expression*)

Description

The COS function returns the cosine of *numeric-expression*. The value of *numeric-expression* is interpreted as radians, degrees, or grads according to the current setting of the angle units (RAD, DEG, or GRAD).

The value returned by the COS function is in the range of −1 through 1 for any angle units you select.

Examples

```
100 RAD:T=COS(PI)
```
Sets T equal to −1 (the cosine of PI radians).

```
200 GRAD:PRINT COS(30):PAUSE
```
Prints .8910065242 (the cosine of 30 grads).

Cross Reference

ACOS, ASIN, ATN, DEG, GRAD, RAD, SIN, TAN

COSH

The COSH function computes the hyperbolic cosine of a numeric expression.

Format COSH(*numeric-expression*)

Description The COSH (hyperbolic cosine) function calculates the hyperbolic cosine of *numeric-expression*. The definition of hyperbolic cosine is shown below.

COSH(X) = .5*(EXP(X) + EXP(-X))

Examples

```
100 PRINT COSH(0) : PAUSE
```
Prints 1.

```
230 T=COSH(0.75)
```
Sets T equal to 1.294683285.

Cross Reference ACOSH, ASINH, ATANH, SINH, TANH

DATA

The DATA statement is used with the READ statement to assign values to variables.

Format DATA *data-list*

Description When a READ statement is executed, the values in *data-list* are assigned to the variables specified in the variable list of the READ statement.

Data-list may consist of one or more numeric or string constants separated by commas. Leading and trailing spaces are ignored. Therefore, a string constant that contains commas or leading or trailing spaces must be enclosed in quotes.

A quotation mark within a string is represented by two adjacent quotation marks. A null string is represented by two adjacent commas, or by two commas separated by two adjacent quotation marks.

If a numeric variable is specified in the variable list of the READ statement, a numeric constant must be in the corresponding position in the *data-list* of the DATA statement.

If a string variable is specified in the READ statement, either a string or a numeric constant may be in the corresponding position in the DATA statement. However, the data will be interpreted as a string.

Using the DATA Statement in a Program

A DATA statement must be the only statement on a line. It may be located anywhere in a program or subprogram. If a program has more than one DATA statement, the statements are normally read in sequential order beginning with the lowest numbered line.

The RESTORE statement can be used to reread DATA statements or to alter the order in which DATA statements are read.

Examples

Notice that the DATA statements can be placed anywhere within the program.

```
100 FOR A=1 TO 5
110 READ B,C
120 PRINT B;C:PAUSE 1.1
130 NEXT A
```

Lines 100 through 130 read five sets of data and print their values, two to a line.

```
140 DATA 2,4,6,7,8
150 DATA 1,2,3,4,5
160 DATA """THIS HAS QUOTES"""
170 DATA NO QUOTES HERE
180 DATA "NO QUOTES HERE, EITHER"
```

Lines 160 and 180 use quotation marks to enclose strings that contain quotation marks and a comma.

```
190 FOR A=1 TO 7
200 READ B$
210 PRINT B$:PAUSE 1.1
220 NEXT A
```

Lines 190 through 220 read seven data elements and print each on its own line.

```
230 DATA 1, NUMBER,,TI
```

Line 230 uses a null string.

Cross Reference READ, RESTORE

DEG

The DEG statement sets the units of angle calculations to degrees.

Format DEG

Description

The DEG statement sets the angle units to degrees until you:

- Enter RAD or GRAD as a command or statement to change the units.
- Enter the NEW ALL command.
- Initialize the system.
- Change the angle setting in CALC mode.

Entering the NEW ALL command or initializing the system automatically changes the angle setting to RAD.

Examples

```
100 DEG
```

Selects the DEG angle setting.

```
200 DEG:PRINT COS(90):PAUSE
```

Prints 0 (the cosine of 90 degrees).

Cross Reference GRAD, RAD

DELETE

The DELETE (or DEL) command removes lines from the program currently in memory or a file from an external storage device.

Formats

DELETE *line-group* [, *line-group* . . .]
DELETE "*device.filename*"

Description

The two formats for the DELETE command are described below. The abbreviation DEL may be used instead of DELETE.

- DEL *line-group* [, *line-group* . . .]—deletes the specified lines from the program in memory. You can specify *line-group* as a single line number or a range of line numbers. You can also specify several *line-groups* by separating them with a comma.

Line-group	Lines Deleted
DEL 100	Line number 100 only
DEL 100–	Lines from number 100 to the highest line number, inclusive
DEL –100	Lines from the lowest line number to line number 100, inclusive
DEL 100–300	Lines from number 100 to number 300, inclusive

If *line-group* specifies only one line number and that line number does not exist, the message `W11 Line number error` is displayed. However, any other listed *line-group* is deleted when you press **[ENTER]**.

If the initial line of a range does not exist, the next higher-numbered line is used as the initial line. If the final line does not exist, the next lower-numbered line is used as the final line.

Description (Continued)

- DEL "*device.filename*"—deletes the file specified by *device.filename*, where *device* identifies the peripheral device on which the file is stored and *filename* identifies the name of the file.

  Device must be a number from 1 through 255 that corresponds to the device number of the peripheral. Refer to the peripheral manual for information concerning the device number.

  Some peripheral devices also enable you to delete data files by using DELETE in the CLOSE statement. Refer to the peripheral manual for more information.

Examples

```
DEL 10-50,90,110-220
```
Deletes lines 10 through 50, 90, and 110 through 220.

```
DEL 900-
```
Deletes lines 900 through the end of the program.

```
DEL -500, 750
```
Deletes all lines through 500 and line 750.

```
DEL "8.inventory"
```
Deletes the file "inventory" from device 8.

Cross Reference

CLOSE, NEW

DIM

The DIM statement defines the dimensions, size, and type of an array.

Format

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

Description

When a DIM statement is executed, the TI-74 chooses a block of memory, labels it with the specified array name, and allocates enough memory space to contain the elements of the array. Note that a DIM statement can define more than one array.

Array-name is a string or numeric variable name. The data contained in an array must be either string or numeric data, as specified by the array name.

The number of values in parentheses following *array-name* determines the number of dimensions in the array. Arrays with up to three dimensions are allowed. Each value represents the maximum subscript in that dimension of the array.

The lowest value of a subscript is zero. Therefore, the number of elements in each dimension is one more than the maximum subscript. For example, an array defined by DIM A(6) is a one-dimensional array with seven elements, A(0) through A(6).

When execution of a program begins, each element of a numeric array is set to zero, and each element of a string array is set to the null string.

If an array is not defined by a DIM statement, the maximum value of each subscript is set to 10 when your program first references the array.

Using a DIM Statement

When you are using a DIM statement in a program, the following rules apply.

- An array can be dimensioned only once.
- To define an array, a DIM statement must be executed before the first reference to the array.
- Remarks (REM) and tail remarks (!) are the only statements which may appear after a DIM statement on a multiple-statement line.
- A DIM statement cannot appear in an IF THEN ELSE statement.

Examples

```
120 DIM X$(30)
```

Reserves space in the computer's memory for 31 elements of the string array called X$. Each element is initialized to the null string.

```
430 DIM D(100),B(10,9)
```

Reserves space in the computer's memory for 101 elements of the array called D and 110 (11 times 10) elements of the array called B. Each element of each array is initialized to zero.

Cross Reference

Refer to "Using Arrays" in Chapter 1.

DISPLAY

The DISPLAY statement displays the value(s) included in *print-list*. (The options available with DISPLAY often make it more versatile than the PRINT statement.)

Formats

DISPLAY [[AT(*column*)] [ERASE ALL] [SIZE(*numeric-expression*)] [USING *line-number*,]]*print-list*

DISPLAY [[AT(*column*)] [ERASE ALL] [SIZE(*numeric-expression*)] [USING *string-expression*,]] *print-list*

Description

The DISPLAY statement formats and displays the value(s) included in *print-list*.

Print-list consists of print items and print separators. A print item can be a numeric expression, a string expression, or a TAB function. A print separator can be a comma or a semicolon, which determines the position of the next print item in the display. For information about the effect of commas and semicolons on spacing, refer to PRINT in this chapter.

Options

The options available with DISPLAY give you control over the format of displayed information. They can be included in any order and must be preceded by a space (unless preceded by the close parenthesis ")" of a previous option).

- AT(*column*)—positions the first character of the displayed information at the column specified by the rounded value of *column*. Valid *column* values are from 1 through 80. If *column* is larger than 80, the TI-74 returns the `E4 Bad value` error message.

  The evaluation of the TAB function and comma separators is relative to the starting position specified by the AT option. However, if the characters to be displayed exceed column 80, the displayed information begins in column 1, not in the column specified by the AT option, the TAB function, or a comma separator.

  The AT option may be affected by the SIZE option. See the following page for additional information.

Options (Continued)

If AT is omitted, output begins in location 1 unless a pending input/output statement exists, and the TAB function and comma separators are relative to column 1.

- ERASE ALL—clears the line before displaying data.
- SIZE(*numeric-expression*)—limits the number of displayed characters to the absolute value of *numeric-expression*.

  If *numeric-expression* is larger than the number of remaining positions, the display field extends from the current display position to the end of the line. The length of the display field defined in this manner becomes the new record length for evaluating the TAB function and comma separators in *print-list*. The display field is cleared prior to displaying data.

  If the SIZE option is omitted and the characters to be displayed exceed column 80, you can only view the information by including a PAUSE ALL statement prior to the ouput line. You can then view the print item one line at a time, pressing **[ENTER]** or **[CLR]** to display the next segment. However, if the SIZE option is used when characters exceed column 80, the text is truncated to 80 columns or to the number of characters specified by SIZE, whichever is shorter.

  When the SIZE option is omitted, the display is not cleared (unless ERASE ALL is specified) prior to displaying data. If there is no trailing separator after *print-list*, termination of the DISPLAY statement clears the display from the last item displayed to the end of the line.

- USING—may be used to specify an exact format for the output. If USING is specified, it must appear last in the option list. Refer to IMAGE and USING for a description of format definition and its effect upon the output of the DISPLAY statement.

DISPLAY (Continued)

Examples

```
120 DISPLAY AT(7),Y
```
Displays the value of Y starting at column 7 and clears everything following the number. The value actually appears in column 8 since the sign precedes the number.

```
150 DISPLAY N
```
Displays the value of N in column 1 of the display and clears the rest of the display.

```
190 DISPLAY ERASE ALL,B
```
Clears the entire display before displaying the value of B.

```
370 DISPLAY AT(C) SIZE(19),X
```
Clears 19 characters starting at position C and displays the value of X starting at position C.

Cross Reference IMAGE, PAUSE, PRINT, TAB, USING

END

The END statement closes all open files and then stops program execution. Although it may appear anywhere, END is often placed as the last line in the main program.

Format END

Description The END statement is not a required statement. Normally, a program stops automatically after the highest-numbered line in the main program is executed.

All subroutines of the main program must occur before the END statement. However, subprograms can be placed after the END statement.

Cross Reference STOP

EOF

The EOF function is used when a program is inputting records from a file. It tests whether there are records remaining in the specified file.

Format

EOF(*file-number*)

Description

The EOF function returns a value that indicates the current position in the file specified by *file-number*. The value of *file-number* must correspond to the number of an open file. Otherwise, the error message `File error` is displayed.

Value	Position
0	Not end-of-file
−1	Logical end-of-file

The logical end-of-file occurs when all records on the file have been input. EOF always treats a file as if it were being accessed sequentially, even if it has been opened for relative access.

Using the EOF Statement

EOF is often placed before an INPUT statement to test the file status before attempting to read from the file. When a program uses pending INPUT statements (refer to INPUT with files), EOF does not indicate whether pending input data remains in the memory buffer.

Examples

```
710 IF EOF(27) THEN 1150
```

Transfers control to line 1150 if the end-of-file has been reached for file #27.

The following statements open a file and check to see if the end-of-file is reached before trying to read a record. When the end-of-file is reached, the file is closed.

```
100 OPEN #3,"2.CFILE",INTERNAL
110 IF EOF(3) THEN CLOSE #3:STOP
120 INPUT #3, A$,D,E
130 PRINT A$;D;E:PAUSE 1
140 GOTO 110
```

Cross Reference

INPUT (with files)

ERR Subprogram

The ERR subprogram returns the error code, error type, and optionally, the file number and line number of the last uncleared error.

Format

CALL ERR(*error-code,error-type* [, *file-number*, *line-number*])

Description

When an error occurs, a subroutine can be called (see ON ERROR) that contains CALL ERR. The error is cleared when this error-processing subroutine terminates with a RETURN.

If no error has occurred, CALL ERR returns all values as zeros.

Parameters

- *error-codes*—range from 0 through 127. The meaning of each error code is listed in Appendix D.
- *error-type*—is always 0 unless *error-code* is 0, which is an input/output (I/O) error. For an I/O error, *error-type* is an I/O error code specified by each I/O device. The range for I/O error codes is 0 through 255.

Options

- *file-number*—is 0 unless the error is an I/O error. For an I/O error, *file-number* is the file number used in the I/O statement that caused the error.
- *line-number*—is the number of the line being executed when the error occurred. It is not always the line that is the source of the problem since an error may occur because of values generated or actions taken elsewhere in a program.

Examples

```
170 CALL ERR(A,B)
```

Sets A equal to the *error-code* and B equal to the *error-type* of the most recent uncleared error.

```
390 CALL ERR(W,X,Y,Z)
```

Sets W equal to the *error-code*, X equal to the *error-type*, Y equal to the *file-number*, and Z equal to the *line-number* of the most recent uncleared error.

Cross Reference

ON ERROR, RETURN (with ON ERROR)

EXP

The EXP function computes the antilogarithm, e^x, of a numeric expression. The internal value for e is 2.71828182846.

Format

EXP(*numeric-expression*)

Description

The EXP function returns the value of e^x, where x equals the value of *numeric-expression*.

EXP is the inverse of the LN function.

Examples

```
150 PRINT EXP(7):PAUSE
```
Prints 1096.633158 (the value of e raised to the 7th power).

```
390 L=EXP(4.394960467)
```
Sets L equal to the value of e raised to the 4.394960467 power, which is 81.04142689.

Cross Reference

LN

FOR TO STEP

The FOR TO STEP and NEXT statements are used to set off a series of statements to be performed a specific number of times.

Format

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

Description

The FOR TO STEP statement is used with the NEXT statement to form a loop, a series of statements performed a specific number of times. *Control-variable* is an unsubscripted numeric variable that acts as a counter for the loop. *Initial-value*, *limit*, and *increment* are numeric expressions.

When the FOR statement is executed, *initial-value* is assigned to *control-variable*. If *initial-value* exceeds *limit*, the loop is skipped and execution continues with the statement after the NEXT statement. Otherwise, the statements following the FOR statement are executed until the corresponding NEXT statement is executed. *Increment* is then added to *control-variable*. If *control-variable* is not greater than *limit*, execution returns to the statement following the FOR statement.

When *control-variable* becomes greater than *limit*, control transfers to the statement following the NEXT statement. *Control-variable* then equals the value it had in the last pass through the loop plus the value of *increment*.

A loop contained entirely within another loop is called a nested loop. Nested loops must use different control variables. Program execution can be transferred out of a loop using GOTO, GOSUB, or IF THEN ELSE and then returned back into the loop.

If a NEXT statement is executed before its corresponding FOR statement, an error occurs.

STEP *increment* specifies the value that is added to *control-variable* each time the loop is executed. If STEP *increment* is omitted, the increment is 1. If *increment* is negative, *control-variable* is decreased each time through the loop and *limit* should be less than *initial-value*. The loop is skipped if *initial-value* is less than *limit*. Otherwise, the loop is executed until *control-variable* is less than *limit*.

Examples

```
140 FOR A=1 TO 5 STEP 2
.
.
.
190 NEXT A
```

Executes the statements between FOR and NEXT A three times, with A having values of 1, 3, and 5. After the loop is finished, A has a value of 7.

```
250 FOR J=7 TO -5 STEP -.5
.
.
.
350 NEXT J
```

Executes the statements between FOR and NEXT J 25 times, with J having values of 7, 6.5, 6, ..., −4, −4.5, and −5. After the loop is finished, J has a value of −5.5.

```
700 FOR X=1 TO 2 STEP -1
.
.
.
780 NEXT X
.
.
.
```

Does not execute the loop because increment is negative and the initial value is already less than the limit.

Cross Reference NEXT

FORMAT

The FORMAT statement initializes a medium on an external storage device.

Format FORMAT *device*

Description

Some external storage devices cannot store on a medium unless the medium has been initialized. The FORMAT statement initializes the medium installed on an external storage device.

Device is the number associated with each physical device and can be from 2 through 255. Refer to the peripheral manuals to obtain the device code for each peripheral device.

Initializing destroys all information previously stored on a medium.

Note: Formatting does not apply to cassette tapes used on a cassette recorder.

Example

```
140 FORMAT 2
```

Initializes the medium currently in the mass-storage device designated as device number 2. All data previously stored on the medium is destroyed.

FRE

The FRE function provides information about the current use of memory in the computer.

Format

FRE(*numeric-expression*)

Description

The FRE function returns information about memory availability and usage:

- How much is currently available for program and data storage.
- How much is occupied by the current program in memory.

The value of *numeric-expression* selects the type of information as follows.

Value	Meaning
0	Memory available for program and data storage (memory space **not** reserved for system operation).
1	Total space occupied by the program currently in memory. The value returned includes 11 bytes for program overhead.

Example

```
300 A=FRE(1)
```
Sets A equal to the number of bytes required to store the current program.

GET Subprogram

The GET subprogram replaces the contents of system memory with information from a RAM cartridge.

Format

CALL GET (*image-number*)

Description

The GET subprogram retrieves a copy of a system RAM image from an 8K Constant Memory cartridge. The term "image" applies to all contents of the 8K system RAM, including program lines, variables, and unused space.

The *image-number* can be 1 or −1. The 1 causes the cartridge image to be copied into memory and the −1 causes an exchange of memory images. This option enables you to store the program from memory while retrieving a cartridge program.

As cartridge contents are copied or exchanged, the computer checks to see that the cartridge contains an image of system memory. If not, the computer returns an error message.

Examples

```
CALL GET(1)
```
Copies the cartridge image into system RAM.

```
CALL GET(-1)
```
Exchanges the cartridge image with the system RAM.

Cross Reference

PUT, ADDMEM

GOSUB

The GOSUB statement stores a return location and then transfers program control to a subroutine.

Format

GOSUB *line-number*

Description

The GOSUB statement transfers control to the subroutine that begins at *line-number*. The statements of the subroutine are executed until a RETURN statement is encountered. A RETURN statement returns control to the statement following the GOSUB statement.

Subroutines may be called any number of times in a program and may call themselves or other subroutines. The GOSUB statement cannot be used to transfer control into or out of a subprogram.

Note: When a subroutine references itself, you must make the GOSUB conditional. Otherwise, the computer's memory can be filled with return addresses.

Example

```
100 GOSUB 200
```
Transfers control to line 200. The statement at line 200 and all the statements that follow are performed until RETURN is encountered.

Cross Reference

ON GOSUB, RETURN

GOTO

The GOTO statement transfers program execution to another line within a program.

Format

GOTO *line-number*

Description

When a GOTO statement is executed, control is passed to the first statement on the line specified by *line-number*.

The GOTO statement cannot be used to transfer control into or out of a subprogram. Attempting to do so results in an `E11 Line number error` message.

Example

```
100 GOTO 300
```
Transfers control to line 300.

GRAD

The GRAD statement sets the units of angle calculations to grads.

Format

GRAD

Description

The GRAD statement sets the angle units to grads until you:

- Enter DEG or RAD as a command or statement to change the units.
- Enter the NEW ALL command.
- Initialize the system.
- Change the angle setting in CALC mode.

Entering the NEW ALL command or initializing the system automatically changes the angle setting to RAD.

Examples

```
100 GRAD
```
Selects the GRAD angle setting.

```
200 GRAD:PRINT COS(100):PAUSE
```
Prints 0 (the cosine of 100 grads).

Cross Reference

DEG, RAD

IF THEN ELSE

The IF THEN ELSE statement performs a choice of actions based on a condition. A true condition causes one action and a false condition causes a different action.

Format

IF *condition* THEN *action1* [ELSE *action2*]

Description

The IF THEN ELSE statement performs one of two specified actions based on a specified condition. If *condition* is true, *action1* is performed. If *condition* is false, *action2* is performed. If ELSE is omitted and *condition* is false, control passes to the next line.

Condition can be either a relational expression or a numeric expression. When a relational expression is evaluated, the result is 0 if it is false and −1 if it is true. When a numeric expression is evaluated, a zero value is considered to be false and a nonzero value is considered to be true.

Action1 and *action2* may be line numbers, statements, or groups of statements separated by colons. If a line number is used, control is transferred to that line. If statements are used, those statements are performed.

The IF THEN ELSE statement must be contained on one line. IF THEN ELSE statements can be nested by including an IF THEN ELSE statement in *action1* or *action2*. If a nested IF THEN ELSE statement does not contain the same number of THEN and ELSE clauses, each ELSE is matched with the closest unmatched THEN.

IF THEN ELSE statements cannot contain DIM, IMAGE, SUB, or SUBEND statements.

Examples

```
140 IF MBB=0 THEN 200
150 PRINT "NON-ZERO" : PAUSE 2
```

If MBB is zero, control passes to line 200. If MBB is not zero, NON-ZERO is displayed and program execution halts for 2 seconds before executing the next statement.

```
230 IF X>5 THEN GOSUB 300 ELSE X=X+5
```

If the value of X is greater than 5, GOSUB 300 is executed. When the subroutine is completed, control returns to the line following the IF THEN ELSE statement. If X is 5 or less, X is set equal to X+5 and control passes to the next line.

```
250 IF Q THEN C=C+1 :GOTO 500 ELSE L=L/C:
    GOTO 300
```

If Q is not zero (true), C is set equal to C+1 and control is transfered to line 500. If Q is zero (false), L is set equal to L/C and control is transferred to line 300.

```
290 IF A$="Y" THEN COUNT=COUNT+1 :
    DISPLAY AT(4),"Enter value:";:GOTO 200
```

If A$ is equal to "Y", COUNT is incremented by 1, a message is displayed, and control is transferred to line 200. If A$ is not equal to "Y", control passes to the next line.

```
350 IF HRS<=40 THEN PAY=HRS*WAGE ELSE
    PAY=HRS*WAGE+.5*WAGE *(HRS-40) :OT=1
```

If HRS is less than or equal to 40, PAY is set equal to HRS*WAGE and control passes to the next line. If HRS is greater than 40, PAY is set equal to HRS*WAGE+.5*WAGE*(HRS-40), OT is set equal to 1, and control passes to the next line.

```
700 IF A=1 THEN IF B=2 THEN C=3 ELSE D=4
```

If A is equal to 1 and B is equal to 2, C is set equal to 3 and control passes to the next line. If A is equal to 1 and B is not equal to 2, D is set equal to 4 and control passes to the next line. If A is not equal to 1, control passes to the next line.

IMAGE

The IMAGE statement enables you to define an output format.

Format

IMAGE *string-constant*

Description

The IMAGE statement specifies an output format for use in DISPLAY USING and PRINT USING statements. The format is used by placing the line number of the IMAGE statement in the USING option of DISPLAY or PRINT (see USING in this chapter).

String-constant may be enclosed in quotation marks. If *string-constant* is not enclosed in quotation marks, leading and trailing blanks are ignored.

The IMAGE statement must be the only statement on a program line and must appear in the program or subprogram that uses it. When an IMAGE statement is encountered, execution immediately continues with the next line of the program.

Format Definition

When a PRINT or DISPLAY statement uses a format definition, the format fields are replaced by the values of the print items, and the literal fields are printed exactly as they appear in the format definition.

The three characters that may be used to define a format field are the number sign (#), the decimal point (.), and the exponentiation symbol (^). The number sign defines a character position in the format field. It is replaced by one of the characters from the ASCII representation of the value of the print item. The decimal point is used in a decimal format field to specify the position of the decimal point. The exponentiation symbol (^) is used in an exponential format field to specify the number of positions in which to print the exponent value.

All other characters are literal and thus form literal fields.

Types of Fields

The five types of fields in a format definition are integer, decimal, exponential, string, and literal. The rules that apply to each type are listed below.

Integer Field

- Up to 14 significant digits may be specified.
- An integer field is composed of number signs.
- When the number does not fill the field, the number is right-justified.
- When the number is longer than the field, asterisks (*) are printed in place of the value.
- Non-integer values are rounded to the nearest integer.
- When the number is negative, one number sign is used for the minus sign.

Decimal Field

- Up to 14 significant digits may be specified.
- A decimal field is composed of number signs and a single decimal point. The decimal point may appear anywhere in the format field.
- The number is placed with the decimal point in the specified position.
- When the integer part of the value is longer than the integer part of the format, asterisks (*) are printed instead of the value.
- The number is rounded to the number of places specified to the right of the decimal point.
- When the number is negative, at least one number sign must precede the decimal point to be used for the minus sign.

Exponential Field

- Up to 14 significant digits may be specified.
- An exponential field consists of a decimal or integer field, which defines the mantissa, followed by 4 or 5 exponentiation symbols that define the exponent. When fewer than 4 are used, they are treated as literal characters. When more than 5 are used, the first 5 are used to define the exponential field, and the remainder are considered to be literal characters.
- The number is rounded according to the mantissa definition.
- When the mantissa definition specifies positions to the left of the decimal point, one of these positions is always used for the sign, minus if negative and a space if positive.

String Field

- The size of the field is limited only by the size of the string that defines the format.
- A string field is an integer, decimal, or exponential field. In addition to the number signs, the decimal point and the exponentiation symbols define character positions.
- When the string is shorter than the field, it is left-justified.
- When the string is longer than the field, asterisks (*) are printed instead of the value.

Literal Field

- The size of the field is limited only by the size of the string that defines the format.
- A literal field is composed of characters that are not format characters. However, decimal points and exponentiation symbols may also appear in literal fields.
- Literal fields appear in the printed output exactly as they appear in the format definition.

Examples

The following program prints two numbers per line using the IMAGE statement.

```
100 FOR COUNT=1 TO 6
110 READ A,B
120 PRINT USING 150; A,B:PAUSE
130 NEXT COUNT
140 DATA 99,-9.99,-7,-3.459,0,0,14.8,12.75,
    79,852,-84,64.7
150 IMAGE THE ANSWERS ARE ## AND ##.##
```

The following chart shows the results.

Values		Displayed Results
99	-9.99	THE ANSWERS ARE 99 AND -9.99
-7	-3.459	THE ANSWERS ARE -7 AND -3.46
0	0	THE ANSWERS ARE 0 AND .00
14.8	12.75	THE ANSWERS ARE 15 AND 12.75
79	852	THE ANSWERS ARE 79 AND *****
-84	64.7	THE ANSWERS ARE ** AND 64.70

Examples (Continued)

The program below illustrates a use of IMAGE. It reads and prints seven numbers and their total. The amounts are printed with the decimal points lined up.

```
100 IMAGE $####.##
110 IMAGE " ####.##"
```

Lines 100 and 110 set up the images. They are the same except for the dollar sign. To keep the blank space where the dollar sign was, the *string-constant* in line 110 is enclosed in quotation marks.

```
120 DATA 233.45,-147.95,8.4,37.263,-51.299,
    85.2,464
130 TOTAL=0
140 FOR A=1 TO 7
150 READ AMOUNT
160 TOTAL=TOTAL+AMOUNT
170 IF A=1 THEN 180 ELSE 190
180 PRINT USING 100,AMOUNT:PAUSE: GOTO 200
190 PRINT USING 110,AMOUNT:PAUSE
```

Prints the values using the IMAGE statements.

```
200 NEXT A
210 PRINT USING "$####.##",TOTAL:PAUSE
```

Uses the format directly in the PRINT statement.

When you run the program, the following values are displayed:

```
$ 233.45
 -147.95
    8.40
   37.26
  -51.30
   85.20
  464.00
$ 629.06  (total amount)
```

Cross Reference DISPLAY, PRINT, USING

INPUT (with keyboard)

The INPUT statement enables you to enter data from the keyboard when you run a program.

Format

INPUT [*input-prompt*;] *variable-list* [, *input-prompt*; *variable-list*] [. . .]

Description

When INPUT is executed, program execution is suspended until you enter data from the keyboard. The entered data is assigned to a variable and execution resumes.

Input-prompt is a string expression that must be followed by a semicolon. If a string constant is used, it must be enclosed in quotes. *Input-prompt* is displayed beginning at the current display position as left by pending input/output statements. If *input-prompt* is omitted, a question mark followed by a space is used for the prompt.

Following the prompt, the flashing cursor is displayed. If the resultant cursor position is greater than 31, the display is cleared and the cursor position is set to column 1 prior to displaying the prompt. When *input-prompt* is greater than 30 characters, it is truncated to 30 characters.

Variable-list is a list of variables separated by commas. The variables may be numeric or string, subscripted or unsubscripted. When more than one variable follows *input-prompt*, the prompt is displayed for the first variable only. Thereafter, the question mark prompt is used until another *input-prompt* is encountered. Each value is assigned to the corresponding variable name before the computer prompts for the next value.

Description (Continued)

When *variable-list* specifies numeric input, you can enter a numeric constant or a numeric expression. The expression is evaluated and the result is assigned to the variable. In string entries, leading and trailing spaces are ignored. Thus, if a string value includes commas, leading spaces, or trailing spaces, you must enclose the string in quotes. Two consecutive quotation marks within a quoted string become a single quotation mark in the displayed string.

If an error occurs during data entry, a descriptive error message is displayed. After the **[ENTER]** or **[CLR]** key is pressed, the INPUT statement reprompts and the data can be entered in the correct form.

When data is entered, the following validations are made.

- If more than one value at a time is entered, the message `E1 at` *line number* `Syntax` is displayed and the data must be reentered one item at a time.
- If a string constant is entered for a numeric variable, the message `E3 at` *line number* `Mismatch` is displayed and a numeric value must be entered.
- If a number whose absolute value is greater than 9.99999999999999E + 127 is entered, the message `W25 at` *line number* `Overflow` is displayed and the value must be reentered.
- If a number whose absolute value is less than 1E – 128 is entered, the value is replaced with 0 and no message is displayed.

Note: When an INPUT statement is waiting for data, **[CLR]** clears only the input field, **[CTL]** **[↑]** (home) and **[CTL]** **[←]** (back tab) move the cursor to the beginning of the input field, and **[CTL]** **[→]** has no effect.

Examples

```
100 INPUT X
```

Causes the computer to display the question-mark prompt and wait for an input value. When **[ENTER]** is pressed, the entered value is stored in the variable X.

```
100 INPUT X$,Y,"ENTER Z";Z(A)
```

Causes the computer to display the question-mark prompt and wait for an input value for X$. When **[ENTER]** is pressed, the entered value is assigned to X$. The question-mark prompt is again displayed and the computer waits for a value to be entered for Y. Then ENTER Z is displayed and the computer waits for an input value for Z(A). The subscript is evaluated for Z(A) before the data value is stored.

Cross Reference ACCEPT, INPUT (with files), LINPUT

INPUT (with files)

The INPUT statement reads data from files that have been opened in INPUT or UPDATE mode.

Format

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

Description

The INPUT statement assigns information in a file to the variables specified in *variable-list*. For INPUT to read a file, it must be opened in INPUT or UPDATE mode.

File-number is a number from 0 through 255 that refers to an open file or device. File number 0 refers to the keyboard and display and is always open. See INPUT (with keyboard). *File-number* is rounded to the nearest integer.

REC *numeric-expression* is used when *file-number* refers to a RELATIVE record file. *Numeric-expression* specifies the record to be read from the file. The first record of a file is record zero. (Refer to individual peripheral manuals for information about RELATIVE record files and the use of the REC clause.) Note that relative files cannot be used with a cassette recorder.

Variable-list is a list of variables separated by commas. The variables may be numeric or string, subscripted or unsubscripted. The data values in the current record are assigned to the variables in the list. If the current record does not contain enough data, another record is read. Successive records are read until each variable is assigned a value or the end-of-file is encountered.

When an INPUT statement terminates, any remaining data values in the current record are ignored. The next INPUT statement that accesses the file reads another record. However, when *variable-list* ends with a comma, the input is left pending. That is, the remaining values in the current record are maintained. The next INPUT statement that accesses the file begins with the next available data value.

If pending input data exists when a PRINT, RESTORE, or CLOSE statement accesses the file, the pending data is discarded. If pending output data exists when an INPUT statement is encountered, the pending data is output before the INPUT statement is executed.

File Types

The computer interprets data differently when reading DISPLAY and INTERNAL files.

DISPLAY-type data has the same form as data entered from the keyboard. The values in each record are separated by commas. Leading and trailing spaces are ignored unless they are part of a string value enclosed in quotation marks. Within a string value enclosed in quotes, two quotation marks represent a single quotation mark. When the INPUT statement encounters two adjacent commas, a null string is assigned to the variable. Each item is verified to ensure that numeric values are placed in numeric variables and string values in string variables.

INTERNAL-type data is in binary format, the format used internally during execution. Each value is preceded by its length. The INPUT statement uses the lengths to separate and assign the values to the variables. The only validation performed by the INPUT statement is to assure that numeric data is from 2 to 8 bytes long.

Examples

```
100 INPUT #1,X$
```
Stores in X$ the next value available in the file that was opened as #1.

```
250 INPUT #23,X,A,LL$
```
Stores in X, A, and LL$ the next three values from the file that was opened as #23.

```
320 INPUT #3,A,B,C,
```
Stores in A, B, and C the next three values from the file that was opened as #3. The comma after C creates a pending input condition.

Examples (Continued)

The following program formats the medium in external device number 2 (thereby destroying any data that was previously on the medium), opens it in update mode, and prints five values to the file MYFILE on the medium. The values are then reread and displayed.

```
100 FORMAT 2
110 OPEN #1,"2.MYFILE",INTERNAL, UPDATE
120 FOR A=1 TO 5
130 READ DATAOUT
140 PRINT #1,DATAOUT
```
Lines 120 through 140 read five records from the DATA statement and write them to file #1.
```
150 PRINT DATAOUT;"IS WRITTEN TO FILE #1.":
    PAUSE 1.5
160 NEXT A
170 RESTORE #1
180 FOR B=1 TO 5
190 INPUT #1,DATAIN
200 PRINT DATAIN;"IS RECORD #";B:PAUSE 1.5
210 NEXT B
```
Lines 180 through 210 read the five records that were written on file #1 and then display their values.
```
220 CLOSE #1, DELETE
```
Deletes the file.
```
230 DATA 15,30,72,36,94
```

Cross Reference

CLOSE, INPUT (with keyboard), LINPUT, OPEN, PRINT, RESTORE

INT

The INT function converts a number into an integer.

Format INT(*numeric-expression*)

Description The INT function returns the largest integer less than or equal to *numeric-expression*.

Examples

```
250 P=INT(3.999999999)
```
Sets P equal to 3.

```
470 DISPLAY AT(7),INT(4.0):PAUSE
```
Displays 4 in column 8.

```
610 K=INT(-3.0000001)
```
Sets K equal to −4.

IO Subprogram

The IO subprogram performs certain operations with peripherals that are not built into TI-74 BASIC.

Format CALL IO (*device,command*[,*status-variable*])

Description The IO subprogram addresses an external device to perform a special control operation not available in TI-74 BASIC. The control operations available with an external device depend on the design of the device.

Proper use of this subprogram requires knowledge of input/output (I/O) data structures and specific peripheral capabilities. Chapter 3 provides information about I/O commands that are available with the PC-324 printer.

Device is the number associated with the external device and can be from 1 through 255.

Command is a numeric code that specifies the operation to be performed by the device.

Status-variable is a numeric variable in which information regarding the result of the operation is stored. If no I/O error occurred, *status-variable* is zero. If an I/O error occurred, *status-variable* contains the corresponding error code.

The inclusion of a *status-variable* affects the computer's response to the occurrence of an I/O error. If an I/O error occurs when *status-variable* is included, no error message is displayed and the error cannot be handled by ON ERROR. If an error occurs when *status-variable* is omitted, the message is displayed or the error can be handled by ON ERROR.

Example

```
140 CALL IO(1,1)
```
Closes device 1. (A command code of 1 is a CLOSE operation.)

Cross Reference ON ERROR

KEY Subprogram

The KEY subprogram enables you to check whether or not a key is being pressed. If a key is pressed, the KEY subprogram detects which key it is.

Format

CALL KEY(*return-variable,status-variable*)

Description

The KEY subprogram scans the keyboard for input and assigns the ASCII code of a key pressed to *return-variable*. If no key is pressed, *return-variable* is set equal to 255. See Appendix B for a list of the ASCII codes.

The value assigned to *status-variable* represents the status of the scan. A value of 0 means no key is pressed. A value of 1 means a different key is pressed since the last time the keyboard was scanned for input (e.g., since CALL KEY, KEY$, INPUT, LINPUT, or ACCEPT was last executed). A value of −1 means the same key is pressed.

Example

The following program segment prompts twice for a key to be pressed. To determine that the responses are distinct, the *status-variable* is compared to 1 (S<>1) in lines 520 and 560.

```
500 PRINT "MORE ENTRIES? (Y OR N)"
510 CALL KEY(K,S)
520 IF S<>1 THEN 510
530 IF K=ASC("Y") OR K=ASC("y") THEN 400
540 PRINT "END SESSION? (Y OR N)"
550 CALL KEY(K,S)
560 IF S<>1 THEN 550
570 IF K=ASC("Y") OR K=ASC("y") THEN STOP
```


KEY$

The KEY$ function enters a one-character string during program execution.

Format

KEY$

Description

The KEY$ function halts program execution until a single key is pressed. When a key is pressed, execution of the program continues immediately and KEY$ returns a one-character string that corresponds to the key pressed. Refer to Appendix B for a list of the ASCII character codes.

If **[BREAK]** is pressed while KEY$ is waiting for a response, the break occurs as usual.

Example

The following program continues if Y is pressed and stops if N is pressed.

```
100 PRINT "Press Y to continue, N to stop"
110 A$=KEY$
120 IF A$="Y" OR A$="y" THEN 140
130 IF A$="N" OR A$="n" THEN 150 ELSE 110
140 PRINT "Continue":PAUSE 1.5 :GOTO 100
150 PRINT "Stop":PAUSE
```

LEN

The LEN function returns a string's length.

Format LEN(*string-expression*)

Description The LEN function returns the number of characters in *string-expression*. A space counts as a character.

Examples

```
170 PRINT LEN("ABCDE"):PAUSE
```
Prints 5.

```
230 X=LEN("THIS IS A SENTENCE.")
```
Sets X equal to 19.

```
540 X$="THIS IS A SENTENCE.":X=LEN(X$)
```
Sets X equal to 19.

```
910 DISPLAY LEN(""):PAUSE
```
Displays 0.

LET

The LET keyword can begin a statement that assigns values to variables.

Formats

[LET] *numeric-variable*[,*numeric-variable* . . .]
= *numeric-expression*

[LET] *string-variable*[,*string-variable*. . .]
= *string-expression*

Description The LET statement assigns the value of an expression to the specified variable(s). The computer evaluates the expression on the right and places the result into the variable(s) on the left.

The keyword LET is optional.

If you list more than one variable, they must be separated with commas.

All subscripts on the left are evaluated before any assignments are made.

Examples

```
110 LET T=4
```
Sets T equal to 4.

```
170 X,Y,Z=12.4
```
Sets X, Y, and Z equal to 12.4.

```
200 A=3<5
```
Sets A equal to −1 because it is true that 3 is less than 5.

```
350 L$,D$,B$="B"
```
Sets L$, D$, and B$ equal to "B".

LINPUT

The LINPUT statement assigns an input string or record to a variable.

Formats

LINPUT [*input-prompt*;]*string-variable*

LINPUT [*#file-number*,[REC *numeric-expression*,]] *string-variable*

Description

The LINPUT statement assigns a string, an entire input record, or the remainder of a pending input record to *string-variable*. Unlike INPUT, LINPUT performs no editing on the input data. Thus, all characters including commas, leading and trailing spaces, semicolons, and quotation marks are placed into *string-variable*.

Input-prompt is a string expression that must be followed by a semicolon. If a string constant is used, it must be enclosed in quotes. *Input-prompt* is displayed beginning at the current display position as left by pending input/output statements. If *input-prompt* is omitted, a question mark followed by a space is used for the prompt.

Following the prompt, the flashing cursor is displayed. If the resultant cursor position is greater than 31, the display is cleared and the cursor position is set to column 1 prior to displaying the prompt. When *input-prompt* is greater than 30 characters, it is truncated to 30 characters.

Using LINPUT with Files

LINPUT can be used to read display-type data from a file or a device. *File-number* is the number of an open file. If the specified file has pending input, the remainder of the pending record is read. The message `E7 Bad data` is displayed if the record or partial record is longer than 255 characters.

The optional REC clause may be used with devices that support relative record (random access) files. *Numeric-expression* specifies the record to be accessed. Refer to the appropriate peripheral manual for more information concerning relative files. Note that relative files cannot be used with a cassette recorder.

Examples

```
300 LINPUT L$
```
Causes the computer to display the question-mark prompt and store the entered data into L$.

```
470 LINPUT "NAME: ";NM$
```
Causes the computer to display NAME: and store the entered data into NM$.

```
470 LINPUT #3,PHONE$
```
Causes the computer to read a record from file #3 and assign the record to PHONE$.

Cross Reference

INPUT, OPEN, CLOSE, PRINT, RESTORE

LIST

The LIST command enables you to view or print program lines.

Formats

LIST [*line-group*]

LIST "*device-name*"[,*line-group*]

Description

The LIST command begins a program listing. If *line-group* is not included, the entire program is listed. When *line-group* is included, only those lines are listed. *Line-group* may specify any of the following line ranges.

Line-group	Effect
100	Lists line 100 only.
100–	Lists line 100 and all following lines.
–300	Lists line 300 and all preceding lines.
100–300	Lists line numbers 100 through 300, inclusive.

When *device-name* is included, the lines are listed to the specified device. If *device-name* is omitted, the lines are shown in the display. During a listing to the display, the lines may be edited.

To suspend a listing to a device, press and hold any key until the listing stops. Pressing the key again resumes the listing. Pressing **[BREAK]** terminates the listing to the display or a device. Pressing [↑] terminates only a listing to the display.

Examples

```
LIST 100-200
```
Lists all lines from 100 through 200 to the display.

```
LIST "12"
```
Lists the entire program to peripheral device 12 (a compatible printer).

```
LIST "50.R=C", -200
```
Lists all lines up to and including line 200 to device 50.

LN

The LN function computes an expression's natural logarithm.

Format

LN(*numeric-expression*)

Description

The LN function returns the natural logarithm of *numeric-expression*. *Numeric-expression* must be greater than zero or the error message `E23 Bad argument` is displayed.

The LN function is the inverse of the EXP function.

Examples

```
710 PRINT LN(3.4):PAUSE
```
Prints the natural logarithm of 3.4, which is `1.223775432`.

```
850 X=LN(EXP(2.7))
```
Sets X equal to the natural logarithm of e raised to the 2.7 power, which equals 2.7.

```
910 S=LN(SQR(T))
```
Sets S equal to the natural logarithm of the square root of the value of T.

Cross Reference

EXP, LOG

LOG

The LOG function computes an expression's common logarithm.

Format LOG(*numeric-expression*)

Description The LOG function returns the common logarithm of *numeric-expression*. *Numeric-expression* must be greater than zero or the error message `E23 Bad argument` is displayed.

Examples

```
150 PRINT LOG(3.4):PAUSE
```
Prints the common logarithm of 3.4, which is `.531478917`.

```
230 S=LOG(SQR(T))
```
Sets S equal to the common logarithm of the square root of the value of T.

Cross Reference LN

NEW

The NEW command prepares the computer for a new program by deleting the program and variables currently in memory.

Formats

NEW
NEW ALL

Description The NEW command closes any open files and then deletes the program and variables currently in memory.

The NEW ALL command performs the following operations:

- Deletes the current program and variables in memory.
- Closes any open files.
- Clears user-assigned strings.
- Cancels any expansion of memory implemented by CALL ADDMEM.
- Sets the angle units to radians.

NEXT

The NEXT statement is used in conjunction with a FOR TO STEP statement to form a loop.

Format

NEXT [*control-variable*]

Description

The NEXT statement increments a *control-variable* during the execution of a FOR/NEXT loop. NEXT also sends execution back to its corresponding FOR TO STEP statement unless *control-variable* is exceeded. A loop that begins with FOR TO STEP must end with NEXT.

If *control-variable* is included, it must be the same as *control-variable* in the FOR TO STEP statement. If *control-variable* is omitted, NEXT is paired with the most recent unmatched FOR TO STEP statement. It is good programming practice to include *control-variable.*

When FOR/NEXT loops are nested, the NEXT statement for the inside loop must appear before the NEXT statement for the outside loop.

See FOR TO STEP for a description of the looping process.

Example

The program below illustrates a use of the NEXT statement. The values printed are 30 and –2.

```
100 TOTAL=0
110 FOR COUNT=10 TO 0 STEP -2
120 TOTAL=TOTAL+COUNT
130 NEXT COUNT
140 PRINT TOTAL;COUNT:PAUSE
```

Cross Reference

FOR TO STEP

NUMBER

The NUMBER (or NUM) command causes automatic line numbering during the entry of program lines.

Format

NUMBER [*initial-line*] [,*increment*]

Description

The NUMBER (or NUM) command generates sequenced line numbers. A line number is displayed with a trailing space for convenience in entering a program line. After you type a line and press **[ENTER]**, the line is stored in memory and the next line number is displayed.

If *initial-line* and *increment* are not specified, the line numbers start at 100 and increase in increments of 10. Otherwise, lines are numbered according to the *initial-line* and *increment* specified. If a generated line number specifies a line that already exists, that line is displayed and may then be replaced or changed using the edit functions. If a generated line number is altered, the sequence of line numbers continues from the new line number.

To terminate the numbering process, press **[ENTER]** when a line comes up with no statements on it, or press **[BREAK]** when any line is displayed.

Examples

`NUM 110`
Instructs the computer to number starting at 110 with increments of 10.

`NUM 105,5`
Instructs the computer to number starting at line 105 with increments of 5.

Cross Reference

RENUMBER

NUMERIC

The NUMERIC function tests whether or not a string expression is a valid representation of a numeric constant. This test enables you to prevent VAL from using an invalid representation of a number.

Format NUMERIC(*string-expression*)

Description The NUMERIC function returns a value of −1 (true) if *string-expression* is a valid numeric constant, and 0 (false) if *string-expression* is not a valid numeric constant.

Leading and trailing blanks in *string-expression* are ignored. NUMERIC can be used to determine if the VAL function can work correctly on a string meant to represent a number.

Example The following program segment determines if an entry from the keyboard is a valid representation of a numeric constant. If not, an error message is displayed until data is reentered. If the data can be represented as a numeric constant, its numeric value is stored in variable A.

```
100 LINPUT "ENTER VALUE: ";A$
110 IF NOT NUMERIC(A$) THEN LINPUT "ERROR,
    REENTER: ";A$:GOTO 110
120 A=VAL(A$)
```

Cross Reference VAL

OLD

The OLD command loads a program from an external device into memory.

Format OLD "*device.file-name*"

Description The OLD command closes all open files, removes the program currently in memory, and loads a stored program. A BASIC program can be stored on *device.file-name* with the SAVE command.

Device.file-name identifies the device where the program is stored and the name of the file. *Device* is the number associated with the physical device and can be from 1 through 255. *File-name* identifies the particular file. Refer to a peripheral manual for the device code for that peripheral device and for specific information about the form of *file-name*. Refer to Chapter 3 for information about using a cassette recorder.

Note: You cannot retrieve information from a cartridge with the OLD command. Also, you cannot load a data file with the OLD command. If *file-name* specifies a data file rather than a program file, it may be necessary to press the **[RESET]** key.

Examples

```
OLD "2.PROGRAM1"
```

Loads the file PROGRAM1 into the computer's memory from peripheral device 2.

```
OLD "1."
```

Loads the next file into the computer's memory from peripheral device 1, the cassette recorder. If the next file is not a program file, the TI-74 displays an error.

Cross Reference GET, INPUT (with files), PUT, SAVE, VERIFY

ON BREAK

The ON BREAK statement determines the action taken when a breakpoint occurs.

Formats

ON BREAK STOP
ON BREAK NEXT
ON BREAK ERROR

Description

The ON BREAK statement sets the computer to respond to a breakpoint according to the option selected.

- ON BREAK STOP selects the normal function of BREAK, which is to halt program execution and display the standard breakpoint message. The RUN command also selects this function of breakpoints.

- ON BREAK NEXT causes breakpoints to be ignored. When a breakpoint that immediately precedes a line number is encountered, the breakpoint is ignored and the program line is executed. The **[BREAK]** key is also ignored. However, a BREAK statement that does not contain a *line-number-list* halts the program even though ON BREAK NEXT is in effect. ON BREAK NEXT can be used to ignore breakpoints that you have specified in a program for debugging purposes.

  Note: Because the **[BREAK]** key is ignored, the **[RESET]** key must be pressed to stop a program that does not stop normally.

- ON BREAK ERROR causes breakpoints to be treated as errors, thus allowing the ON ERROR statement to process breakpoints. See ON ERROR for more information.

The ON BREAK statement remains in effect until another ON BREAK statement changes it. When a subprogram ends, the ON BREAK status in effect when the subprogram was called is again in effect.

Example

The program below illustrates the use of ON BREAK. When the message `W29 at 120 Break` is displayed, resume execution with the CON command.

```
100 BREAK 140
```
Sets a breakpoint in line 140.
```
110 ON BREAK NEXT
```
Sets breakpoint handling to ignore breakpoints.
```
120 BREAK
```
A breakpoint occurs in line 120 in spite of line 110. Press **[CLR]**, type **CON**, and press **[ENTER]**.
```
130 FOR A=1 TO 500
140 PRINT "(BREAK) IS DISABLED"
150 NEXT A
```
The **[BREAK]** key does not work while lines 130 through 150 are being executed.
```
160 ON BREAK STOP
```
Restores the normal use of **[BREAK]**.
```
170 FOR A=1 TO 50
180 PRINT "NOW (BREAK) WORKS"
190 NEXT A
```
The **[BREAK]** key again works while lines 170 through 190 are being executed.

Cross Reference

BREAK, ON ERROR

ON ERROR

The ON ERROR statement determines the action taken when an error occurs during the execution of a program.

Formats

ON ERROR STOP
ON ERROR *line-number*

Description

After the ON ERROR statement is executed, any errors that occur are handled according to the option selected.

- The ON ERROR STOP statement selects the normal way of handling errors, which is to halt program execution and print a descriptive error message. The RUN command also selects this way of handling errors.

- The ON ERROR *line-number* statement transfers control to the specified line when an error occurs. *Line-number* must be the beginning of an error-processing subroutine. Once an error has occurred and control has been transferred, error handling reverts to ON ERROR STOP. If the ON BREAK ERROR option was selected, it is changed to ON BREAK NEXT. For an error-processing subroutine to handle any new errors, an ON ERROR *line-number* must be executed again.

The ON ERROR statement remains in effect until another ON ERROR statement changes it. If a subprogram ends, and no errors occurred while the subprogram was executing, the ON ERROR status in effect when the subprogram was called is again in effect. If an error occurred in a subprogram, any changes in the error-handling status made by the error handler is in effect when the subprogram ends.

The main program and subprograms can share the same error-processing subroutine, which is called by means of the *line-number* in the ON ERROR statment. The main program and subprograms cannot share subroutines called by GOSUB.

Example

The program below illustrates the use of ON ERROR.

```
100 ON ERROR 150
```
Causes any error to pass control to line 150.
```
110 X$="A"
120 X=VAL(X$)
```
Causes an error.
```
130 PRINT X;"SQUARED IS";X*X:PAUSE 2
140 STOP
150 REM ERROR SUBROUTINE
160 ON ERROR 220
```
Causes the **next** error to pass control to line 220.
```
170 CALL ERR(CODE,TYPE,FILE,LINE)
```
Determines the error using CALL ERR.
```
180 IF LINE<>120 THEN RETURN 220
```
Transfers control to line 220 if the error is not in the expected line.
```
190 IF CODE<>23 THEN RETURN 220
```
Transfers control to line 220 if the error is not the one expected.
```
200 X$="5"
```
Changes the value of X$ to an acceptable value.
```
210 RETURN
```
Returns control to the line in which the error occured.
```
220 REM UNKNOWN ERROR
230 PRINT "ERROR";CODE;" IN LINE";LINE:PAUSE
```
Reports the nature of the unexpected error and the program stops.

Cross Reference

ON BREAK, ON WARNING, RETURN (with ON ERROR)

ON GOSUB

The ON GOSUB statement sends execution to a choice of subroutines.

Format

ON *numeric-expression* GOSUB *line-number1* [,*line-number2*] [, . . .]

Description

The ON GOSUB statement determines which subroutine to execute by evaluating *numeric-expression*.

- If the value of *numeric-expression* is 1, the subroutine starting at *line-number1* is executed; if 2, the subroutine starting at *line-number2* is executed, and so forth.
- If *numeric-expression* is a non-integer, it is rounded.
- If *numeric-expression* is zero, negative, or larger than the list of line numbers, the error message `E4 Bad value` is displayed.

Each line number must be the first statement of a subroutine. After the RETURN statement of the subroutine is executed, control returns to the statement following ON GOSUB. ON GOSUB may not be used to transfer control into or out of a subprogram.

Examples

```
140 ON X GOSUB 1000,2000,300
```
Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
240 ON P-4 GOSUB 200,250,300,800,170
```
Transfers control to 200 if P – 4 is 1 (P is 5), 250 if P – 4 is 2, 300 if P – 4 is 3, 800 if P – 4 is 4, and 170 if P – 4 is 5.

Cross Reference

GOSUB, RETURN (with GOSUB)

ON GOTO

The ON GOTO statement sends execution to a choice of lines within a program.

Format

ON *numeric-expression* GOTO *line-number1* [,*line-number2*] [, . . .]

Description

The ON GOTO statement determines where to transfer control by evaluating *numeric-expression*.

- If the value of *numeric-expression* is 1, control is transferred to *line-number1*; if 2, control is transferred to *line-number2*, and so forth.
- If *numeric-expression* is a non-integer, it is rounded.
- If *numeric-expression* is zero, negative, or greater than the list of line numbers, the error message `E4 Bad value` is displayed.

ON GOTO may not be used to transfer control into or out of a subprogram.

Examples

```
130 ON X GOTO 1000,2000,300
```
Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
210 ON P-4 GOTO 200,250,300,800,170
```
Transfers control to 200 if P – 4 is 1 (P is 5), 250 if P – 4 is 2, 300 if P – 4 is 3, 800 if P – 4 is 4, and 170 if P – 4 is 5.

Cross Reference

GOTO

ON WARNING

The ON WARNING statement determines the effect a warning has during program execution.

Formats

ON WARNING PRINT
ON WARNING NEXT
ON WARNING ERROR

Description

The ON WARNING statement sets the computer to respond to a warning according to the option selected.

- ON WARNING PRINT selects the normal use of warnings, which is to print a descriptive warning message and continue program execution after the **[ENTER]** or **[CLR]** key is pressed.The RUN command also selects this option of ON WARNING.
- ON WARNING NEXT causes the program to continue execution without printing any message.
- ON WARNING ERROR causes the occurrence of a warning to be treated as an error, thus allowing the ON ERROR statement to process warnings.

The ON WARNING statement remains in effect until another ON WARNING statement changes it. When a subprogram ends, the ON WARNING status in effect when the subprogram was called is again in effect.

Example

The program below illustrates the use of ON WARNING.

```
100 ON WARNING NEXT
```
Sets warning handling to go to the next statement.
```
110 PRINT 110,5/0:PAUSE
```
Prints the result without any message.
```
120 ON WARNING PRINT
```
Sets warning handling to the normal option, which is to print a message and enable execution to continue when a warning occurs.
```
130 PRINT 130,5/0:PAUSE
```
Prints the warning. When **[ENTER]** or **[CLR]** is pressed, prints 130 followed by the value of 5/0.
```
140 ON WARNING ERROR
```
Sets warning handling to treat warnings as errors.
```
150 PRINT 150,5/0:PAUSE
```
Prints the warning message and treats the warning as an error. Program execution stops.

Note: When you clear the error condition, the display is cleared. However, you can press **[SHIFT] [PB]** to see the printed value 150.

Cross Reference

ON ERROR

OPEN

The OPEN statement sets up a link to a peripheral device for the purpose of transferring data.

Format

OPEN *#file-number*,"*device.file-name*"[,*file-organization*] [,*file-type*] [,*record-length*] [,*open-mode*]

Description

The OPEN statement enables a BASIC program to use data files and peripheral devices by providing a link between *file-number* and a file or device. In setting up this link, the OPEN statement specifies how the file or device can be used (for input or output) and how the file is organized.

The OPEN statement must be executed before any BASIC statement in a program attempts to use a file or device requiring a file number. If an OPEN statement references a file that already exists, the *file-organization*, *file-type*, and *record-length* attributes in the OPEN statement must be the same as those attributes of the file. If an OPEN statement references a file that is already open, an error occurs.

File-number is a number from 1 through 255 that the OPEN statement associates with a file or device. This *file-number* is used by all the input/output statements that access the file or device. *File-number* is rounded to the nearest integer. File number 0 is the keyboard and display of the computer. It cannot be used for other files and is always open.

Device.file-name is an actual peripheral device number and other device-dependent information. *Device.file-name* may be a string expression. *Device* is the number associated with the physical device and can be from 1 through 255. *File-name* supplies information to the peripheral device for the OPEN statement. For example, with an external storage device, *file-name* specifies the name of the file. With other devices, *file-name* specifies options such as parity, data rate, etc. Refer to the peripheral manuals for the device code for each peripheral device and for specific information about the form of *file-name*.

File Attributes

The file attributes listed below may be in any order or may be omitted. If an attribute is omitted, defaults are used.

File-organization specifies either a sequential or relative (random access) file. Records in a sequential file are read or written in sequence from beginning to end. Records in a RELATIVE (or random access) file can be read or written in any record order, including sequentially. The default *file-organization* is sequential; therefore, omit *file-organization* to indicate sequential files, or specify RELATIVE for random-access files.

File-type may be either DISPLAY or INTERNAL. DISPLAY specifies that the data is written in ASCII format. INTERNAL specifies that the data is written in binary format. Binary records take up less space, are processed more quickly by the computer, and are more efficient for recording data on external storage devices. However, if the information is going to be printed or displayed for people to read, DISPLAY format should be used. If *file-type* is omitted, DISPLAY is assumed.

Record-length consists of the word VARIABLE followed by a numeric expression that specifies the maximum record length for the file. The maximum record length is dependent on the device used. If *record-length* is omitted, the peripheral device specifies a default record length.

Open-mode instructs the computer to process the file in UPDATE, INPUT, OUTPUT, or APPEND mode. UPDATE specifies that data may be both read from and written to the file. INPUT specifies that data may only be read from the file. OUTPUT specifies that data may only be written to the file. APPEND specifies that data may only be written at the end of the file. If *open-mode* is omitted, UPDATE is assumed.

Note that if a file already exists on external storage, specifying OUTPUT mode results in new data being written over the existing data.

Opening Files to a Cassette Tape Recorder

The cassette interface is designated as device 1 in an OPEN statement. The attributes for a cassette recorder file are as follows.

- Files must be sequential.
- The default record length is 256 bytes.
- Files must have an *open-mode* of INPUT or OUTPUT.
- Files must be DISPLAY *file-type*.

Note that you cannot use the RESTORE or DELETE commands, the UPDATE or APPEND *open-mode*, or INTERNAL *file-type* with files on a cassette recorder.

Refer to Chapter 3 for instructions on using a cassette recorder for storage and retrieval of files.

Examples

```
100 OPEN #23,"2.X",INTERNAL,UPDATE
```
Opens the file named "X" on peripheral device 2 and enables any input/output statement to access the file by using the number 23. The type of the file is INTERNAL. Because the file is opened in UPDATE mode, data can be both read from and written to the file.

```
150 OPEN #243,A$&".ABC",INTERNAL
```
If A$ equals "2", opens a file on device 2 with a name of ABC. The file type is INTERNAL, UPDATE mode is assumed, and the device specifies the default record length.

```
200 OPEN #1,"1.DATA1",DISPLAY,OUTPUT
```
Opens the file named "DATA1" on a cassette recorder. The type of the file is DISPLAY. Because the file is opened in OUTPUT mode, data can only be written to the file.

Cross Reference

CLOSE, DELETE, INPUT, LINPUT, PRINT, RESTORE

PAUSE

The PAUSE statement allows displayed information to stay in the display by suspending program execution either for a fixed duration or indefinitely.

Formats

PAUSE *numeric-expression*
PAUSE
PAUSE ALL

Description

The PAUSE statement suspends program execution either for a specified number of seconds or until the **[CLR]** or **[ENTER]** key is pressed. The three forms of the PAUSE statement are described below.

- PAUSE *numeric-expression* suspends program execution for the number of seconds represented by the absolute value of *numeric-expression*. If *numeric-expression* is positive, the timed pause can be overridden by pressing the **[ENTER]** or **[CLR]** key. If *numeric-expression* is negative, the timed pause can only be overridden by pressing the **[BREAK]** key.

  A timed pause is performed in tenths of a second. If *numeric-expression* is less than .1, the program does not pause. During a timed pause, the cursor is not displayed and the display cannot be scrolled.

- PAUSE (without *numeric-expression* or ALL) performs an indefinite pause. The underline cursor is displayed in column one to indicate an indefinite pause is occurring. The cursor control keys can then be used to view the contents of the 80-column line. Execution continues when either **[ENTER]** or **[CLR]** is pressed.

- The PAUSE ALL statement suspends program execution each time a complete output line is sent to the display. Execution continues when **[CLR]** or **[ENTER]** is pressed. PAUSE ALL remains in effect until a timed PAUSE of length zero is executed.

  PAUSE ALL remains in effect when a subprogram is called. If the subprogram includes a PAUSE 0 statement, PAUSE ALL is again in effect when the subprogram ends.

Examples

```
120 PAUSE 2.2
```
Halts execution for 2.2 seconds or until the **[CLR]** or **[ENTER]** key is pressed.

```
190 PAUSE
```
Halts execution until the **[CLR]** or **[ENTER]** key is pressed.

The following program changes degrees Fahrenheit to degrees Celsius.

```
100 PRINT "ENTER DEG: ";
```
Prints the prompt ENTER DEG: . The pending print, created by the semicolon at the end of the PRINT statement, causes the prompt to be displayed until data is entered.

```
110 ACCEPT DG
120 PRINT DG;"DEG =";(DG-32)*5/9;"DEGREES C":
    PAUSE
```
Prints the answer. The PAUSE statement that follows the PRINT statement causes the answer to be displayed until the **[ENTER]** or **[CLR]** key is pressed.

```
130 GOTO 100
```

The following program demonstrates the effect of PAUSE ALL.

```
100 PAUSE ALL
110 PRINT "FIRST PAUSE":PAUSE 1.1
```
Displays FIRST PAUSE until the **[ENTER]** or **[CLR]** key is pressed. Note that the timed pause is performed after you press **[ENTER]** or **[CLR]**.

```
120 PRINT "SECOND PAUSE"
```
Displays SECOND PAUSE until the **[ENTER]** or **[CLR]** key is pressed.

```
130 PAUSE 0:PRINT "THIRD PAUSE":PAUSE .9
```
Cancels the automatic pause and displays THIRD PAUSE for approximately .9 seconds.

Cross Reference DISPLAY, PRINT

PI

PI represents the numeric constant π, the ratio of a circle's circumference to its diameter.

Format PI

Description The PI function returns the value of π accurate to 13 digits, 3.141592653590.

Example

```
130 VOLUME=4/3*PI*R ^ 3
```
Sets VOLUME equal to four-thirds times PI times the radius cubed, which is the volume of a sphere with a radius of R.

POS

The POS function computes the starting position of a string contained within another string.

Format

POS(*string1*, *string2*, *numeric-expression*)

Description

The POS function returns the starting position of the first occurrence of *string2* in *string1*. The search begins at the position specified by *numeric-expression*. If no match is found, the function returns a value of zero.

Examples

```
110 X=POS("PAN","A",1)
```
Sets X equal to 2 because A is the second letter in PAN.

```
140 Y=POS("APAN","A",2)
```
Sets Y equal to 3 because the A in the third position in APAN is the first occurrence of A in the portion of APAN that was searched.

```
170 Z=POS("PAN","A",3)
```
Sets Z equal to 0 because A was not in the part of PAN that was searched.

```
290 R=POS("PABNAN","AN",1)
```
Sets R equal to 5 because the first occurence of AN starts with the A in the fifth position in PABNAN.

PRINT (with display)

This version of the PRINT statement places information in the display.

Formats

PRINT [USING *line-number*,] [*print-list*]
PRINT [USING *string-expression*,] [*print-list*]

Description

The PRINT statement is used to format and write data to the display with the following options.

- USING—may be used to specify an exact format for the items in *print-list*. Refer to IMAGE and USING for a description of format definition and its effect on the output of the PRINT statement.
- *Print-list*—consists of print items and print separators. A print item can be a numeric expression, a string expression, or a TAB function. A print separator can be a comma or a semicolon, which determines the position of the next print item in the display. If *print-list* is omitted, the PRINT statement clears the display.

Print Items

When a PRINT statement is executed, the values of the expressions in *print-list* are displayed in order from left to right in the positions determined by the print separators and TAB functions.

- A string expression is evaluated to produce a string result. A string constant must be enclosed in quotation marks. Blank spaces are not inserted before or after a string. To print a blank space before or after a string, include it in the string or insert it separately with quotes.
- A numeric expression is evaluated and displayed with a trailing space. Positive values are printed with a leading space (instead of a plus sign), and negative numbers are printed with a leading minus sign.
- The TAB function specifies the starting position for the next item in *print-list*. See TAB for more information.

Print Items (Continued)

If a print item is longer than the remainder of the current line, the line is cleared and the print item is displayed starting in column 1. If a numeric print item fits on the current line without its trailing space, it is printed on the current line. If a print item is longer than 80 characters, you can only view the information by including a PAUSE ALL statement prior to the ouput line. You can then view the print item one line at a time, pressing **[ENTER]** or **[CLR]** to display the next segment.

Print Separators

You must place at least one separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon prints the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma prints the next print item at the beginning of the next print field. The first five print fields are 15 characters long, beginning in columns 1, 16, 31, 46, and 61. The last print field begins in column 76, completing the 80-column line. If the current column position is past the start of the last print field, the comma clears the line and displays the next printed item starting at column 1.

Pending Print Conditions

Using a comma or a semicolon after *print-list* creates a pending print condition. A pending print condition allows information from a subsequent input/output statement to be printed on the current line. If a comma ends the PRINT statement, the computer spaces over to the start of the next field for the next print item. If a semicolon ends the statement, the computer starts the next print item at the next position unless the subsequent input/output statement changes the position.

Pending Print Conditions (Continued)

A pending print condition can be used to create an input prompt for the ACCEPT, INPUT (with display), or LINPUT statement. The next INPUT or LINPUT statement places its prompt after the pending print item. See ACCEPT, INPUT (with display), and LINPUT for more information.

If *print-list* is not followed by a comma or a semicolon, the line is cleared when a subsequent input/output statement is executed. Therefore, the print items of the next input/output statement begin in column 1.

Numeric Formats

Numbers are printed in either normal decimal form or scientific notation. Scientific notation is used for very small or very large numbers.

When a number is printed in normal decimal form, the following conventions are observed.

- Integers are printed without a decimal point.
- Non-integers are printed with a decimal point. Trailing zeros in the fractional part are omitted. If the number has more than ten significant digits, the value is rounded to ten digits.
- A number whose absolute value is less than one is printed without a zero to the left of the decimal point.

A number printed in scientific notation is in the following form.

mantissa E *exponent*

Numeric Formats (Continued)

When a number is printed in scientific notation, the following conventions are observed.

- The mantissa is printed with seven or fewer digits with one digit always to the left of the decimal.
- Trailing zeros are omitted in the fractional part of the mantissa.
- The exponent is displayed with a plus or minus sign followed by a two- or three-digit exponent.
- When the exponent is two digits, the mantissa is limited to seven digits. When the exponent is three digits, the mantissa is limited to six digits. When necessary, the mantissa is rounded to the appropriate number of digits.

Examples

```
100 PRINT
```
Prints a blank line.

```
210 PRINT "THE ANSWER IS";ANSWER:PAUSE
```
Prints THE ANSWER IS immediately followed by the value of ANSWER.

```
320 PRINT X,Y/2:PAUSE
```
Prints the value of X and in the next field the value of Y/2.

```
450 PRINT "NAME: ";
460 ACCEPT N$
```
Prints NAME: and accepts the entry after the prompt.

Cross Reference

ACCEPT, DISPLAY, IMAGE, INPUT, LINPUT, PAUSE, TAB, USING

PRINT (with files)

The PRINT statement is used with files to send data to a file or a device.

Formats

PRINT *#file-number*[,REC *numeric-expression*] [,USING *line-number*] [,*print-list*]

PRINT *#file-number*[,REC *numeric-expression*] [,USING *string-expression*] [,*print-list*]

Description

The PRINT statement may be used to format and write data to a file or device.

File-number is a number from 0 through 255 that refers to an open file or device. The file must have been opened in OUTPUT, UPDATE, or APPEND mode. *File-number* 0 refers to the display, which is always open. *File-number* is rounded to the nearest integer.

Options

The following options may be used in the PRINT statement.

- REC *numeric-expression*—may be included only when *file-number* refers to a RELATIVE record file. Refer to the peripheral manuals for information about RELATIVE record files and the proper use of REC. *Numeric-expression* is evaluated to designate the specific record number of the file to which to write.
- USING—specifies an exact format for a DISPLAY-type file. Refer to the IMAGE and USING sections for a description of format definition and its effect upon the PRINT statement. Including USING in a reference to an INTERNAL-type data file results in an error.
- *Print-list*—consists of print items and print separators. Print items are numeric and string expressions and TAB functions. Print separators are commas or semicolons that indicate the position of print items in the record.

  Print-list is interpreted in order from left to right. The form of the output depends upon the type (DISPLAY or INTERNAL) of the file or device. See OPEN for a description of *file-type*.

Options (Continued)

When *print-list* is omitted and there is no pending record, the result depends upon the file type. If the file is DISPLAY-type, the PRINT statement writes a blank (zero length) record. If the file is INTERNAL-type, an error occurs because INTERNAL-type files do not support zero length records.

DISPLAY-type Files

During execution of a PRINT statement that refers to a DISPLAY-type file, *print-list* is evaluated as follows.

- String expressions are evaluated to produce a string result. String constants must be enclosed in quotation marks. Blank spaces are not inserted before or after a string. To print a blank space before or after a string, include it in the string or insert it separately with quotes.
- Numeric expressions are evaluated and displayed with a trailing space. Positive values are printed with a leading space (instead of a plus sign) and negative numbers are printed with a leading minus sign.
- The TAB function specifies the starting position in the record for the next item in *print-list*. See TAB for more information.

You must place at least one print separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon writes the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma writes the next print item at the beginning of the next print field. The print fields are 15 characters long and are located at columns 1, 16, 31, and so forth. If the current column position is past the start of the last print field, the comma causes the next printed item to be printed in the next record.

DISPLAY-type Files (Continued)

If a print item is longer than the remainder of the current record, the current record is written without that item, and the print item is written at the start of the next record. If a numeric print item fits in the current record without its trailing space, it is written in the current record. If a print item is longer than the record length, it is divided into segments that are the length of the record until the last segment is the length of the record or less. The segments are then written in successive records.

INTERNAL-type Files

During execution of a PRINT statement that refers to an INTERNAL-type file or device, *print-list* is evaluated as follows.

- String expressions are evaluated and written in the record in internal string representation.
- Numeric expressions are evaluated and written in the record in internal numeric representation.
- The TAB function causes an error.

You must place at least one print separator between adjacent print items. Multiple print separators in a PRINT statement are evaluated from left to right.

- The semicolon writes the next print item immediately after the last print item, with no extra spaces between the values.
- The comma functions exactly the same as the semicolon separator.

If a print item is longer than the remainder of the current record, the current record is written without that item, and the print item is written at the start of the next record. If a print item is longer than the record length, an error occurs.

Pending Print Conditions

Using a comma or a semicolon after *print-list* creates a pending print condition. If the *print-list* ends with a comma or semicolon, the current record is not written. If a comma ended the PRINT statement, the computer advances to the start of the next field. If a semicolon ended the statement, the next output statement that accesses this file writes data on this same record, beginning at the current position unless the statement changes the position.

When *print-list* is omitted and there is a pending output record, the PRINT statement writes the pending record.

If *print-list* ends without a comma or a semicolon, the record is immediately written to the file. The next input/output statement that accesses the file begins a new record.

Examples

```
150 PRINT #32,A,B,C,
```

Causes the values of A, B, and C to be written to the next record of the file that was opened as number 32. The final comma creates a pending print condition. The next PRINT statement accessing file #32 is written to the same record as this PRINT statement.

The program below writes data to a file.

```
100 OPEN #5,"1.MYPROG",DISPLAY,OUTPUT
```

Opens file number 5. MYPROG is created on the cassette tape in the recorder.

```
110 DIM A(50)
```

Dimensions an array for 51 values.

```
120 B=0
```

Initializes the summation variable.

```
130 FOR J=0 TO 50
```

Lines 130 through 180 facilitate data input.

```
140 PRINT "ENTER VALUE";
150 ACCEPT A(J)
160 B=B+A(J)
170 PRINT #5, A(J);
```

Value of A(J) is written to the file.

```
180 NEXT J
190 PRINT #5,B
```

Value of summation variable is written to the the file.

```
200 CLOSE #5
```

Cross Reference

IMAGE, INPUT (with files), OPEN, TAB, USING

PUT Subprogram

The PUT subprogram sends the contents of system memory to a RAM cartridge.

Format

CALL PUT (*image-number*)

Description

The PUT subprogram stores a copy of the system RAM image in an 8K Constant Memory cartridge. The term "image" applies to all contents of the 8K system RAM, including program lines, variables, and unused space.

The *image-number* can be 1 or −1. The 1 causes the memory image to be copied into the cartridge and the −1 causes an exchange of memory images. This option enables you to store the program from memory while retrieving a cartridge program.

As the cartridge contents are exchanged with memory, the computer checks to see that the cartridge contains an image of system memory. If not, the computer returns an error message.

Examples

```
CALL PUT(1)
```
Copies the system RAM image into the cartridge.

```
CALL PUT(-1)
```
Exchanges the cartridge image with the system RAM.

Cross Reference

GET, ADDMEM

RAD

The RAD statement sets the units of angle calculations to radians.

Format

RAD

Description

The RAD statement sets the angle units to radians until you:

- Enter DEG or RAD as a command or statement to change the units.
- Change the angle setting in CALC mode.

Entering the NEW ALL command or initializing the system automatically sets the angle units to RAD.

Examples

```
100 RAD
```
Selects the RAD angle setting.

```
200 RAD:PRINT COS(PI/2):PAUSE
```
Prints 0 (the cosine of $\pi/2$ radians).

Cross Reference

DEG, GRAD

RANDOMIZE

The RANDOMIZE statement randomizes the sequence of random numbers generated by the RND function.

Format

RANDOMIZE [*numeric-expression*]

Description

The RANDOMIZE statement sets the random number generator to an unpredictable sequence.

If RANDOMIZE is followed by *numeric-expression*, the same sequence of random numbers is produced each time the statement is executed with that value. Different values produce different sequences.

Example

The program below accepts a value for *numeric-expression* and prints the first 10 random numbers obtained using the RND function. Press **[BREAK]** to stop the program.

```
100 INPUT "SEED: ";S
110 RANDOMIZE S
120 FOR A=1 TO 10:PRINT A;RND:PAUSE 1.1
130 NEXT A
140 GOTO 100
```

Cross Reference

RND

READ

The READ statement is used with the DATA statement to assign values to variables.

Format

READ *variable-list*

Description

The READ statement assigns a constant listed in a DATA statement to the corresponding variable in *variable-list*.

Variable-list consists of string and numeric variables, either subscripted or unsubscripted, separated by commas. The value from the DATA statement must be the same type as the variable to which it is assigned in READ. Note that a number listed in a DATA statement can be read into a string variable. When two adjacent commas are encountered in the data list, a null string is read.

A single READ statement may read from more than one DATA statement, and several READ statements may read from a single DATA statement.

- The READ statement begins reading from the first DATA statement in the current program or subprogram and proceeds to the next DATA statement when the current data list has been read.
- If a READ statement does not read all of the current data list, the next READ statement begins with the first unread item in the list.
- An attempt to read data after all the data in the current program or subprogram has been read results in an error.

The RESTORE statement can be used to alter the order in which DATA statements are read.

READ can read data only from a DATA statement that is in the same program or subprogram as the READ statement. Each time a subprogram is called, data is read from the first DATA statement.

Cross Reference

DATA, RESTORE

REM

The REM statement makes the rest of a program line into an explanatory remark.

Formats

REM [*character-string*]

![*character-string*]

Description

The REM statement enables you to enter explanatory remarks into your program. Remarks may include any type of information, but they usually explain a section of a program. Although remarks are not executed, they do take up space in memory.

Character-string may include any displayable character. Any character that follows REM, including the statement separator symbol (:) is considered part of the remark. Therefore, if REM is part of a multiple-statement line, it must be the last statement on the line.

The exclamation point (!) is called a tail remark symbol and may be used instead of the word REM. The exclamation point can appear as the first statement on a line or after the last statement in a multiple-statement line. If the exclamation point appears after a statement, the statement separator (:) is not needed. Using the tail remark symbol saves space in the listed form of the program.

Examples

```
150 REM BEGIN SUBROUTINE
```
Identifies a section beginning a subroutine.

```
270 SUBTOTAL=L+B !Calculate subtotal
```
Identifies statements that perform a specific calculation.

RENUMBER

The RENUMBER command enables you to change the numbers of program lines.

Format

RENUMBER [*initial-line*] [,*increment*]

Description

The RENUMBER (or REN) command changes the line numbers of a program. If no *initial-line* is provided, the renumbering starts with 100. If no *increment* is included, an *increment* of 10 is used.

REN also changes all references to line numbers to match the renumbered lines. If a statement refers to a line number that does not exist, the program is renumbered, but a warning is displayed and the line number reference is replaced with 32767, which is not a valid line number.

If the values entered for *initial-line* and *increment* result in the creation of line numbers larger than 32766, the program is left unchanged and the message `E11 Line number error` is displayed.

Examples

```
REN
```
Renumbers all lines to start with 100 and increment by 10.

```
REN ,100
```
Renumbers all lines to start with 100 and increment by 100.

```
REN 10000,5
```
Renumbers all lines to start with 10000 and increment by 5.

Cross Reference

NUMBER

RESTORE

The RESTORE statement changes the order in which data is read from DATA statements or from a file.

Formats

RESTORE [*line-number*]

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

Description

RESTORE specifies that the next READ statement executed accesses the first item in the DATA statement specified by *line-number*.

Line-number must be in the same program or subprogram as the RESTORE statement. If no *line-number* is included, the DATA statement with the lowest numbered line in the current program or subprogram is used. If *line-number* is not a DATA statement, the next DATA statement following it is used.

RESTORE *#file-number* positions that file to the first record. The next input/output statement that refers to *file-number* accesses the first record in the file. Any pending output data is written to the file before the RESTORE statement is executed. Any pending input data is ignored. *File-number* 0 refers to the keyboard and display; therefore RESTORE #0 performs exactly like RESTORE as described above.

Note: The RESTORE statement cannot be used with files stored by a cassette recorder.

REC may be used with devices that support RELATIVE record (random-access) files. *Numeric-expression* specifies the record to which the random-access file is positioned. The next input/output statement that refers to that file accesses that record. Refer to an appropriate peripheral manual for information about RELATIVE files.

Note: The first record of a file is record zero.

Examples

```
150 RESTORE
```
Selects the first DATA statement in the program as the next DATA statement to be read.

```
200 RESTORE 130
```
Selects the DATA statement at line 130 as the next DATA statement to be read. If line 130 is not a DATA statement, the next DATA statement after line 130 is selected.

```
230 RESTORE #1
```
Sets file #1 to the first record in the file, which is record 0.

Cross Reference

DATA, READ

RETURN (with GOSUB)

RETURN ends execution of a subroutine and then returns program control to the line following the subroutine call.

Format RETURN

Description Used with GOSUB, RETURN transfers control back to the statement following the GOSUB or ON GOSUB statement that was last executed.

A subroutine may contain more than one RETURN statement.

Cross Reference GOSUB, ON GOSUB

RETURN (with ON ERROR)

RETURN is used with ON ERROR to end an error-processing subroutine.

Formats
RETURN
RETURN NEXT
RETURN *line-number*

Description Used with ON ERROR, RETURN ends an error-processing subroutine. An error-processing subroutine is called when an error occurs after an ON ERROR *line-number* statement has been executed. The error-processing subroutine can contain any BASIC statements, including another ON ERROR statement.

RETURN with no option transfers control to the statement in which the error occurred, and the statement is executed again.

RETURN NEXT transfers control to the statement following the one in which the error occurred.

RETURN *line-number* transfers control to the line specified. The specified line must be in the same program or subprogram as the error-processing subroutine, even though the error may have occurred in some other subprogram.

RETURN (with ON ERROR) (Continued)

Example

The program below illustrates the use of RETURN with ON ERROR.

```
100 ON ERROR 150
```
Transfers control to line 150 when an error occurs.
```
110 X=VAL("D")
```
Causes an error, so control is transferred to line 150.
```
120 PRINT "DONE":PAUSE 2
```
Prints DONE.
```
130 STOP
140 REM ERROR HANDLING
150 IF A>4 THEN 200
```
Checks to see if the error has occurred four times and transfers control to 200 if it has.
```
160 A=A+1
```
Increments the error counter by one.
```
170 PRINT A;"ERROR(S)":PAUSE 2
```
Prints the number of errors that have occurred.
```
180 ON ERROR 150
```
Because of the error, ON ERROR STOP was selected. Line 180 resets the error handling to transfer to line 150.
```
190 RETURN
```
Returns to the line that caused the error and executes it again.
```
200 PRINT "LAST ERROR":PAUSE 2:RETURN NEXT
```
Is executed only after the error has occurred five times. Prints LAST ERROR and returns to the line following the one that caused the error.

Cross Reference ON ERROR

RND

The RND function generates a pseudo-random number.

Format RND

Description The RND function returns the next pseudo-random number in the current sequence of pseudo-random numbers. The number returned is greater than or equal to zero and less than one.

Unless a RANDOMIZE statement is used, RND generates the same sequence of numbers each time a program is run.

Example
```
10 PRINT 10*RND:PAUSE
```
Prints a random number greater than or equal to 0 and less than 10.

Cross Reference RANDOMIZE

RPT$

The RPT$ function forms a new string by repeating a starting string.

Format

RPT$(*string-expression,numeric-expression*)

Description

The RPT$ function returns a string that is *numeric-expression* repetitions of *string-expression.*

If RPT$ produces a string longer than 255 characters, the excess characters are discarded, and the warning message `W28 Truncation` is displayed.

Examples

```
100 M$=RPT$("ABCD",4)
```
Sets M$ equal to "ABCDABCDABCDABCD".

```
100 PRINT USING RPT$("#",40);X$:PAUSE
```
Prints the value of X$ using an image that consists of 40 number signs.

RUN

The RUN statement can be used to start execution of a program or to retrieve and execute a program with one command.

Formats

RUN [*line-number*]
RUN "*program-name*"
RUN "*device.file-name*"

Description

The RUN statement starts execution of a program.

RUN *line-number* starts execution of the program in memory at the specified *line-number*. Entering RUN without *line-number* starts execution of the program currently in memory beginning with the lowest numbered line.

RUN "*program-name*" searches a software cartridge and starts execution of *program-name* when it is found. If *program-name* is not found or refers to a subprogram, an error occurs. A string expression may be used to specify *program-name.*

RUN "*device.file-name*" deletes the program currently in memory, loads the contents of *file-name* from *device* into memory, and executes it. A string expression may be used to specify *device.file-name.*

Note: If an I/O error occurs, the program currently in memory may be unaffected. Also, if *file-name* specifies a data file rather than a program file, it may be necessary to press the **[RESET]** key.

Before a program is executed, the following process takes place.

- Numeric variables are set to zero, string variables are set to null strings, and all open files are closed.
- Certain errors, such as a FOR statement without a NEXT statement or a line reference out of range, are detected.
- ON BREAK STOP, ON WARNING PRINT, and ON ERROR STOP are selected.

Examples

RUN
Causes the computer to begin execution of the program in memory, starting with the lowest numbered line.

RUN 200
Causes the computer to begin execution of the program in memory starting at line 200.

RUN "1.PRG3"
Causes the computer to load and begin execution of the program in file PRG3 on device 1.

RUN "MAT"
Executes the program MAT (matrices) in the Mathematics software cartridge.

The program below illustrates the use of the RUN statement to execute a program from a program. A menu is created to enable the person using the program to choose what other program to run. The other programs should run this program rather than ending in the usual way, so that the menu is displayed again after they are finished.

```
100 PRINT "Enter 1, 2, or 3 for programs":
    PAUSE 2
110 PRINT "... or enter 4 to stop":PAUSE 2
120 INPUT "YOUR CHOICE: ";C
130 IF C=1 THEN RUN "1.PRG1"
140 IF C=2 THEN RUN "1.PRG2"
150 IF C=3 THEN RUN "1.PRG3"
160 IF C=4 THEN STOP
170 GOTO 100
```


SAVE

The SAVE command stores a BASIC program on an external device.

Format

SAVE "*device.file-name*"[,PROTECTED]

Description

The SAVE command sends a copy of the BASIC program in memory to an external device. By using the OLD command, you can later recall the program into memory.

Before storing the program, SAVE removes any variables from the system that are not used in the program.

Device.file-name identifies the device where the program is to be stored and the file name. *Device* is the number associated with the physical device and can be from 1 through 255. *File-name* identifies the file that contains the program.

When PROTECTED is specified, the program in memory is left unprotected, but the copy on the external storage device is saved in protected format. A protected program cannot be listed, edited, or saved.

Note: You cannot store information in a cartridge with the SAVE command.

Examples

SAVE "1.PRG1"
Saves the program in memory to device 1 under the name PRG1.

SAVE "2.PRG2",PROTECTED
Saves the program in memory to device 2 under the name PRG2. The program may be loaded into memory and run, but it may not be edited, listed, or resaved.

Cross Reference

GET, OLD, PRINT (with files), PUT, VERIFY

SEG$

The SEG$ function forms a string that is a portion of another string.

Format SEG$(*string-expression,position,length*)

Description The SEG$ function returns a substring of a string. The string returned starts at *position* in *string-expression* and extends for *length* characters.

If *position* is beyond the end of *string-expression*, the null string ("") is returned. If *length* extends beyond the end of *string-expression*, only the characters through the end are returned.

Examples

```
100 X$=SEG$("FIRSTNAME LASTNAME",1,9)
```
Sets X$ equal to "FIRSTNAME".

```
200 Y$=SEG$("FIRSTNAME LASTNAME",11,8)
```
Sets Y$ equal to "LASTNAME".

```
240 Z$=SEG$("FIRSTNAME LASTNAME",10,1)
```
Sets Z$ equal to " ".

```
280 PRINT SEG$(A$,B,C):PAUSE
```
Prints the substring of A$ starting at character B and extending for C characters.

SGN

The SGN function enables you to detect whether a value is positive, zero, or negative.

Format SGN(*numeric-expression*)

Description The SGN function returns the mathematical signum function. If *numeric-expression* is positive, a value of 1 is returned. If it is zero, 0 is returned, and if it is negative, −1 is returned.

Examples

```
140 IF SGN(A)=1 THEN 300 ELSE 400
```
Transfers control to line 300 if A is positive and to line 400 if A is zero or negative.

```
790 ON SGN(X)+2 GOTO 200,300,400
```
Transfers control to line 200 if X is negative, line 300 if X is zero, and line 400 if X is positive.

Cross Reference ABS

SIN

The SIN function computes the trigonometric sine of an expression.

Format SIN(*numeric-expression*)

Description The SIN function returns the trigonometric sine of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current setting of angle units.

Example
```
150 DEG:PRINT SIN(3*21.5+4):PAUSE
```
Sets angle units to degrees and prints .930417568.

Cross Reference ACOS, ASIN, ATN, COS, DEG, RAD, GRAD, TAN

SINH

The SINH function computes the hyperbolic sine of an expression.

Format SINH(*numeric-expression*)

Description The SINH (hyperbolic sine) function calculates the hyperbolic sine of *numeric-expression*. The definition of hyperbolic sine is shown below.

SINH(X) = .5*(EXP(X) – EXP(–X))

Examples
```
100 PRINT SINH(0):PAUSE
```
Prints 0.

```
230 T=SINH(0.75)
```
Sets T equal to .8223167319.

Cross Reference ACOSH, ASINH, ATANH, COSH, TANH

SQR

The SQR function computes the square root of an expression.

Format SQR(*numeric-expression*)

Description The SQR function returns the positive square root of *numeric-expression*. SQR(X) is equivalent to X^(1/2).

Numeric-expression cannot be a negative number.

Examples

```
150 PRINT SQR(4):PAUSE
```
Prints 2.

```
780 X=SQR(2.57E5)
```
Sets X equal to the square root of 257,000, which is 506.9516742.

STOP

The STOP statement stops program execution.

Format STOP

Description The STOP statement stops program execution. It can be used interchangeably with the END statement except that STOP may not be placed after subprograms.

Example The program below illustrates the use of the STOP statement. The program adds the numbers from 1 to 100.

```
100 TOT=0
110 NUMB=1
120 TOT=TOT+NUMB
130 NUMB=NUMB+1
140 IF NUMB>100 THEN PRINT TOT:PAUSE 2:STOP
150 GOTO 120
```

Cross Reference END

STR$

The STR$ function converts a numeric value into a string.

Format

STR$(*numeric-expression*)

Description

The STR$ function returns the string representation of the value of *numeric-expression*. No leading or trailing spaces are included.

The STR$ function is the inverse of the VAL function.

Examples

```
150 NUM$=STR$(78.6)
```
Sets NUM$ equal to "78.6".

```
220 LL$=STR$(3E15)
```
Sets LL$ equal to "3.E+15".

```
330 J$=STR$(A*4)
```
Sets J$ equal to a string equal to the value obtained when A is multiplied by 4. For instance, if A is equal to −8, J$ is set equal to "−32".

Cross Reference

NUMERIC, VAL

SUB

The SUB statement labels the beginning of a subprogram.

Format

SUB *subprogram-name*[(*parameter-list*)]

Description

The SUB statement is the first statement in a subprogram and must be the first statement on the line.

A subprogram is a group of statements separated from the main program and accessed by a CALL statement. A subprogram is an efficient way to handle a task that is repeated several times in a program.

Subprogram-name consists of 1 to 15 characters. The first character must be an alphabetic character or an underline. The remaining characters may be alphabetic, numeric, or underline characters. The CALL statement searches for subprograms in a specific order (see CALL for the order) and executes the first subprogram found with *subprogram-name*. If the name of one of your subprograms is the same as a built-in subprogram, the built-in subprogram is executed.

Parameter-list receives the information passed to the subprogram through the *argument-list* of the CALL statement. A parameter may be a simple string variable, a simple numeric variable, or an array. An array is listed as a parameter by writing the array name followed by parentheses. A one-dimensional array is written as A( ), a two-dimensional array as A(,), and a three-dimensional array as A(,,).

The arguments of *argument-list* and the parameters of *parameter-list* need not have the same names. However, the number and the types of arguments in *argument-list* must match the number and types of parameters in *parameter-list* of the SUB statement.

A subprogram terminates when a SUBEXIT or SUBEND statement is executed. Control is returned to the statement following the CALL statement.

Passing Data to a Subprogram

Information is passed to a subprogram either by reference or by value.

If an argument is passed by reference, the subprogram uses the same variables as the calling program. If the value of the variable is changed in the subprogram, the value is also changed in the calling program. A simple variable, an element of an array, or an array listed in *argument-list* is passed by reference. Arrays are always passed by reference.

If an argument is passed by value, only the value of the variable is passed to a variable in *parameter-list*. If the value of the variable is changed in the subprogram, it does not affect the variable in the calling program. Any type of expression in *argument-list* is evaluated and passed by value to the subprogram. Simple variables may be passed by value by enclosing them in parentheses.

Any variables used in a subprogram other than those in *parameter-list* are local to that subprogram, so the same variable names may be used in the main program and in other subprograms. Changing the values of local variables in a program or subprogram does not affect the values of local variables in any other program or subprogram.

Any local variables in the subprogram are initialized each time the subprogram is called.

Using the SUB Statement

Subprograms appear after the main program. If a SUB statement is encountered in a main program, it terminates as if a STOP statement had been executed. Only remarks and END statements may appear between the SUBEND of one program and the SUB of the next subprogram.

The ON BREAK, ON WARNING, ON ERROR, and PAUSE ALL statements in effect when a CALL is executed remain in effect while the subprogram is executing. If the subprogram changes any of these settings, they are changed back when the subprogram terminates.

A subprogram cannot contain another subprogram or share any subroutines except error-processing subroutines.

Examples

```
100 SUB MENU
```
Marks the beginning of a subprogram. No parameters are passed or returned.

```
220 SUB MENU(COUNT,CHOICE)
```
Marks the beginning of a subprogram. The variables COUNT and CHOICE can be used and may change value in the subprogram. If so, their corresponding arguments in the calling statement are changed.

```
330 SUB PAYCHECK(DATE,Q,SSN,PAYRATE,TABLE(,))
```
Marks the beginning of a subprogram. The variables DATE, Q, SSN, PAYRATE, and the array TABLE with two dimensions can be used and may change value in the subprogram. If so, their corresponding arguments in the calling statement are changed. However, if the corresponding argument of DATE, Q, SSN, or PAYRATE is enclosed in parentheses in the CALL statement, the value of that argument cannot be changed. The corresponding array argument of TABLE is passed by reference in the CALL statement and therefore any of its values can be changed by the subprogram.

Cross Reference

CALL, ON BREAK, ON ERROR, ON WARNING, RETURN, SUBEND, SUBEXIT

SUBEND

The SUBEND statement ends a subprogram and returns execution to the line after the subprogram was called.

Format

SUBEND

Description

The SUBEND statement marks the end of a subprogram. When SUBEND is executed, control is passed to the line following the statement that called the subprogram.

The SUBEND statement must always be the last statement in a subprogram and cannot be in an IF THEN ELSE statement.

Only remarks and END statements may appear between a SUBEND statement and the next SUB statement.

Cross Reference

SUB, SUBEXIT

SUBEXIT

The SUBEXIT statement transfers execution out of a subprogram.

Format

SUBEXIT

Description

The SUBEXIT statement terminates execution of a subprogram and transfers control to the line following the statement that called the subprogram.

The SUBEXIT statement may appear as many times as needed in a subprogram.

Cross Reference

SUB, SUBEND

TAB

The TAB function is used in a PRINT or DISPLAY statement to select a specific position for a print item.

Format

TAB(*numeric-expression*)

Description

The TAB function, in conjunction with a PRINT or DISPLAY statement, selects a specific position for a print item.

- If *numeric-expression* is greater than the current position, the TAB function advances to the specified position.
- If *numeric-expression* is less than the current position, the TAB function proceeds to the next record and advances to the specified position.
- If *numeric-expression* is greater than the length of a record for the device being used, then *numeric-expression* is repeatedly reduced by the record length until it is less than the record length.
- If *numeric-expression* is less than or equal to zero, the position is set to 1.

The TAB function is treated as a print item and must be separated from other print items by a print separator. The print separator before TAB is evaluated before the TAB function, and the print separator following TAB is evaluated after the TAB function. Normally, semicolons are used before and after TAB.

In a DISPLAY statement, the TAB function is relative to the beginning of the display field. If AT is used, the TAB function is relative to the specified column position. If displayed information exceeds 80 characters, the TAB function is performed relative to column 1.

If SIZE is used in a DISPLAY statement, the value specified in SIZE is the absolute limit of the number of characters displayed. This limit is the record length used in evaluating any TAB functions.

Examples

```
100 PRINT TAB(12);35:PAUSE
```
Prints the number 35 starting at column 13.

```
190 PRINT 356;TAB(18);"NAME":PAUSE
```
Prints 356 at the beginning of the line and NAME starting at column 18.

```
710 DISPLAY AT(10) SIZE(20),"MGB";TAB(10);
    "ADDR":PAUSE
```
Prints MGB starting at column 10 and ADDR starting at column 19.

Cross Reference

DISPLAY, PRINT (with display), PRINT (with files)

TAN

The TAN function computes the trigonometric tangent of an expression.

Format TAN(*numeric-expression*)

Description The TAN (tangent) function returns the trigonometric tangent of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current angle setting.

Example
```
250 RAD:PRINT TAN(20):PAUSE
```
Sets angle units to radians and prints 2.237160944.

Cross Reference ACOS, ASIN, ATN, COS, DEG, RAD, GRAD, SIN

TANH

The TANH function computes the hyperbolic tangent of an expression.

Format TANH(*numeric-expression*)

Description The TANH function returns the hyperbolic tangent of *numeric-expression*. The definition of hyperbolic tangent is shown below.

$$TANH(X) = (EXP(X) - EXP(-X))/(EXP(X) + EXP(-X))$$

Examples
```
100 PRINT TANH(0):PAUSE
```
Prints 0.

```
230 T=TANH(0.75)
```
Sets T equal to .6351489524.

Cross Reference ACOSH, ASINH, ATANH, COSH, SINH

UNBREAK

The UNBREAK statement removes any breakpoints previously set.

Format

UNBREAK [*line-list*]

Description

The UNBREAK statement removes all breakpoints. If *line-list* is specified, only the breakpoints for those lines are removed.

Examples

```
UNBREAK
```
Removes all breakpoints.

```
400 UNBREAK 100,130
```
Removes the breakpoints set for lines 100 and 130.

Cross Reference

BREAK

USING

USING enables you to define an image that formats output with DISPLAY and PRINT.

Formats

DISPLAY USING *line-number*
DISPLAY USING *string-expression*
PRINT USING *line-number*
PRINT USING *string-expression*

Description

USING is used in a DISPLAY or PRINT statement to format the output.

If *line-number* is included, the format is specified by an IMAGE statement in the referenced line. *Line-number* must refer to a line in the current program or subprogram.

If *string-expression* is included, the format is defined by the string expression.

The Effects of USING with a Print-list

When USING is present, the following changes occur in the evaluation of the *print-list* of PRINT or DISPLAY.

- Comma print separators are treated as semicolons.
- The TAB function causes an error.
- The print items are formatted according to fields specified in the format definition. If the number of print items in *print-list* exceeds the number of fields in the format, the current formatted record is written. The remaining values are written in the next record, using the format definition again, from the beginning. The format is used as many times as is necessary to complete the *print-list*. A new record is generated each time the format is used. When the number of print items is less than the number of fields in the definition, output stops when the first field is encountered for which there is no print item.
- If a formatted item is too long for the remainder of the current record, it is divided into segments. The first segment fills the remainder of the current record and any remaining segments are written on the next record.

Cross Reference

DISPLAY, IMAGE, PRINT

VAL

The VAL function converts a string into a number, provided the string can be read as a number.

Format VAL(*string-expression*)

Description The VAL function returns the numeric value of *string-expression* if it is a valid representation of a number. Leading and trailing spaces are ignored.

If *string-expression* is not a valid representation of a number, an error occurs. To avoid this error, the *string-expression* may be verified first with the NUMERIC function.

The VAL function is the inverse of the STR$ function.

Examples

```
170 NUMB=VAL("78.6")
```
Sets NUMB equal to 78.6.

```
190 LL=VAL("3E15")
```
Sets LL equal to 3.E + 15 (scientific notation).

```
300 PRINT VAL("$3.50"):PAUSE
```
Causes an error because the string contains a non-numeric character ($).

Cross Reference NUMERIC, STR$

VERIFY

The VERIFY command checks that a copy of a program saved on an external storage device or loaded into memory is the same as the original program.

Format VERIFY "*device.file-name*"[,PROTECTED]

Description The VERIFY command compares a file stored externally with the same file in memory. The comparison can be made after a SAVE or OLD command to check the file on the external storage device.

Device.file-name identifies the device and the file in which the program is stored. *Device* is the number associated with the physical device and can be from 1 through 255. *File-name* identifies the file.

If the two files are identical, the program has been stored or retrieved successfully. If a difference is found in the comparison, either I/O error 12 or 24 is displayed.

Like SAVE, VERIFY removes any variable names that are not used in the program. If the program was saved with the PROTECTED option, then PROTECTED must also be specified in the VERIFY command.

Examples

```
SAVE "1.PROGRAM1"
```
Saves the file named PROGRAM1 to device 1.

```
VERIFY "1.PROGRAM1"
```
Verifies whether the file was stored correctly.

```
OLD "1.STAT"
```
Reads the file named STAT into memory from device 1.

```
VERIFY "1.STAT"
```
Verifies whether the file was read correctly.

Cross Reference OLD, SAVE

Chapter 3: Using Optional Accessories

This chapter provides information to help you use the optional CI-7 cassette interface cable, PC-324 printer, and 8K Constant Memory cartridge accessories. With optional equipment, you can save programs and data by recording them on cassette tape, you can get printed results, or you can expand the memory.

Table of Contents

Guidelines for Selecting Equipment

The CI-7 cassette interface cable, a cassette recorder, and a cassette tape become a complete information storage system for the TI-74. The performance of the storage system depends on the type of recorder and cassette tape you select. You may want to consider the following guidelines when you choose a cassette recorder and supplies.

Selecting a Recorder

The TI-74 is compatible with most standard cassette recorders. However, for the best results, choose a cassette recorder with these features:

- Volume control
- Tone Control
- Microphone jack
- Remote jack
- Earphone or external speaker jack
- Digital tape counter (This enables you to quickly locate a specific tape position when you store more than one program or set of data on the same tape.)
- Optional AC adapter (This avoids problems that weak batteries may cause, such as interfering with the transfer of information.)

Selecting Cassette Tapes

For maximum storage capability, follow these guidelines when you select cassette tapes.

- Choose tapes with low-noise characteristics. Tapes with extended frequency response, such as digital tape, are unnecessary and cost more than common audio cassettes.
- Choose quality cassette tapes. Low-quality tapes are prone to tangling and breakage.
- Use the type and length of tape recommended by the manufacturer of your recorder.

Caring for Your Equipment

The following suggestions can help you care for your cassette interface cable, cassette recorder, and cassette tapes.

Caring for the Interface Cable

Several things should be taken into consideration in caring for the cassette interface cable.

- Handle the cable carefully.
- Store the cable in a clean, dry place, protected from high temperatures.

Caring for the Recorder and Cassette Tapes

You should follow several simple guidelines in caring for your cassette recorder and cassette tapes. These suggestions include:

- Clean and demagnetize the tape head periodically because tapes tend to deposit magnetic particles on the tape head that hinder clear information transfer.
- Store your cassette tapes away from magnetic sources, such as a television set, an electric motor, magnetic cabinet latches, magnets in children's toys, and magnetic note holders.

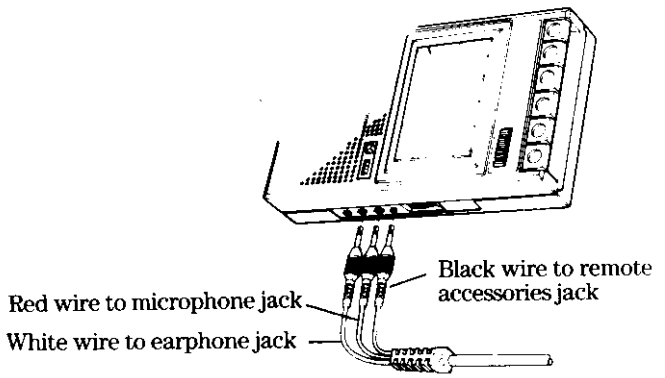
Connecting Your Recorder to the TI-74

The CI-7 cassette interface cable enables the TI-74 to exchange information with the cassette recorder. Only a few simple steps are required to properly connect the interface cable to the TI-74 and your cassette recorder.

Connecting the Recorder

Be sure the TI-74 is off before you connect or disconnect the cable. To connect your recorder to the TI-74:

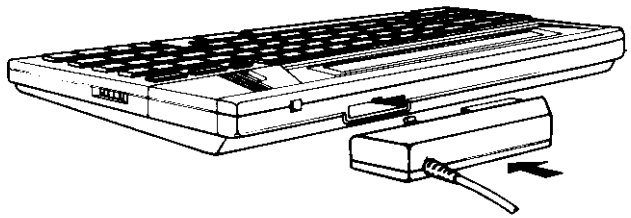
1. Insert the plug on the red wire into the microphone jack of the cassette recorder (usually marked MIC).



2. Insert the plug on the white wire into the jack for the earphone, monitor, or external speaker on the cassette recorder (usually marked EAR or MONITOR).

3. Insert the plug on the black wire into the remote control jack of your cassette recorder (usually marked REM).

4. Plug the other end of the cassette interface cable into the peripheral port on the back of the TI-74.



Prompts for Using the Cassette Recorder

While saving and retrieving programs or data, you must manually operate the cassette recorder. Prompts appear in the display to help you operate the recorder and to tell you what is happening in the recording process.

Automatic Display Prompts

In the recording process, two types of prompts appear in the display.

- Instructional prompts—Tell you what to do to manually operate the recorder and the TI-74. For example, the prompt `Position tape; then press ENTER` tells you what to do at this point in the operation.

- Informational prompts—Tell you what is currently taking place in the recording process. For example, the prompt `Reading...` indicates that the TI-74 is retrieving information from a cassette file.

Recording Without Prompts

After you become familiar with recording operations, you may want to speed up the process by eliminating the automatic display prompts. To disable the automatic prompts, type the suffix ".NM" (for no messages) at the end of the filename. If you disable the prompts before you begin an input or output operation, be sure that the tape is positioned correctly.

For example, if you type **RUN "1.TEST.NM"**, press **[ENTER]**, and then press the PLAY button of your recorder, the TI-74 searches the tape for the file named TEST. When the file is found, you can load and execute the program without receiving any prompts.

Determining the Recorder Settings

Before you begin recording your programs and data files, you need to complete the following test program. Making a test recording establishes the right volume and tone setting for your recorder, checks the compatibility of your recorder and tape, and helps you become familiar with recording a program file.

Procedure	Comments
1. Adjust the volume to a medium setting and, if your recorder has a tone control, adjust the tone to a medium-high setting (7 on a scale of 10).	These settings for volume and tone control work for most recorders. You may need to vary the settings slightly on your recorder. (Refer to page 3-9 for additional information.)
2. Insert a cassette tape into the recorder. Rewind the tape and set the tape counter to zero. Then advance the tape past any leader (004 or higher).	To record information, the tape must be advanced past any leader. The leader is the non-magnetic segment at each end of a cassette tape.
3. Make sure that none of the recorder buttons are depressed. Then turn on the TI-74.	
4. Be sure the TI-74 is in the BASIC mode. Enter the test program: **100 A$ = "TEST"** **110 FOR B = 1 TO 3** **120 FOR C = 1 TO 4** **130 DISPLAY SEG$(A$,1,C):PAUSE .3** **140 NEXT C:NEXT B:END**	
5. Type **SAVE "1.TESTPROG"** and press **[ENTER]**.	Addresses device 1 (the recorder) and assigns a name to this test recording. The I/O indicator appears in the display.

Procedure	Comments
6. When `Position tape; then press ENTER` appears, make sure the tape is past any leader. Note the tape counter setting and press **[ENTER]**.	The tape counter setting helps you locate the beginning of the file when you are ready to retrieve it.
7. When `Press RECORD; then ENTER` appears, press the record button(s) on the recorder and press **[ENTER]**.	After a few seconds, the recorder begins to record the file and the prompt `Writing...` appears.
8. When `Press STOP; then ENTER`, appears, press the STOP button on the recorder. Note the tape counter setting and press **[ENTER]**.	Your file is saved. The TI-74 is ready to store another file or play back what you recorded. You now know the tape counter setting for the beginning and ending of the file.
9. Type **VERIFY "1.TESTPROG"** and press **[ENTER]**.	
10. When `Position tape; then press ENTER` appears, use the buttons on the recorder to locate the beginning of the file. Then press **[ENTER]**.	The file is ready to be read.
11. When `Press PLAY; then ENTER` appears, press the play button on the recorder and press **[ENTER]**.	After a few seconds, the recorder plays back the file and the prompt `Reading...` appears.

Procedure	Comments
12. View the prompt and follow the instruction that applies to you.	
► If `Press STOP; then ENTER` appears, press the STOP button on the recorder. The file is saved to cassette tape.	Your test program is verified. Note the volume and tone settings that were successful for the test recording.
► If `E0 I/O error 3 "1"` appears, refer to "Adjusting the Tone and Volume" on the next page.	The TI-74 could not find the test program. You may have tried to record on the leader, the volume setting may be incorrect, or you may not have started at the beginning of the file.
► If `E0 I/O error 23 "1"` appears, refer to "Adjusting the Tone and Volume" on the next page.	The file is there, but the TI-74 cannot read it because of a technical problem. The volume setting may be incorrect, or the recorder may be incompatible.
► If no error message appears after a considerable length of time and the TI-74 continues to search for the file, check for low batteries. Then go to step 5.	Low batteries may be interfering with the transfer of information. Replace batteries as needed, or use an AC adapter with the recorder.

Adjusting the Tone and Volume

If an error message appears, rewind the tape to the beginning of the file, and increase the volume approximately 20 percent. Return to step 9 of the recording procedure, and repeat the VERIFY command.

If an error message appears again, adjust the volume to approximately 20 percent below the medium setting and repeat the VERIFY command again.

A volume setting between medium-low and medium-high (3–8) works well for most recorders.

If you have tried to verify the recording on several volume settings without success, return the volume control to a medium setting, raise the tone setting approximately 20 percent, and repeat the procedure.

A tone setting between medium and high (5–10) works well for most recorders.

Comments on Volume and Tone

Notice that the volume setting necessary for saving a file may be higher than for normal listening. If you use the recorder for audio purposes, be sure to readjust the volume and tone settings before you record data from the TI-74.

If you use a tape that has different magnetic characteristics from the tape you were previously using, you may need to repeat the procedure for determining the proper volume and tone settings.

Guidelines for Good Recording

Saving files by recording them on cassette tape provides a permanent record of your programs and data. However, several factors can influence the quality of your recordings. The suggestions below can help you record your programs and data efficiently and avoid some potential problems.

Using the Tape Counter

The tape counter enables you to quickly locate a specific point on the tape where a file may be located. To use the tape counter, follow these steps before you begin recording:

1. Rewind the tape to the beginning.
2. Set the tape counter to zero by pressing the tape counter button on your recorder.
3. Advance the tape to the area the file is to occupy and note the number on the counter.

   To avoid false counter readings, do not reset the tape counter unless the tape is rewound to the beginning.
4. Assign each file its own name. Make a note of the file name and the counter positions where the file begins and ends.

Tips for Recording

To avoid problems when recording programs or data, follow these guidelines:

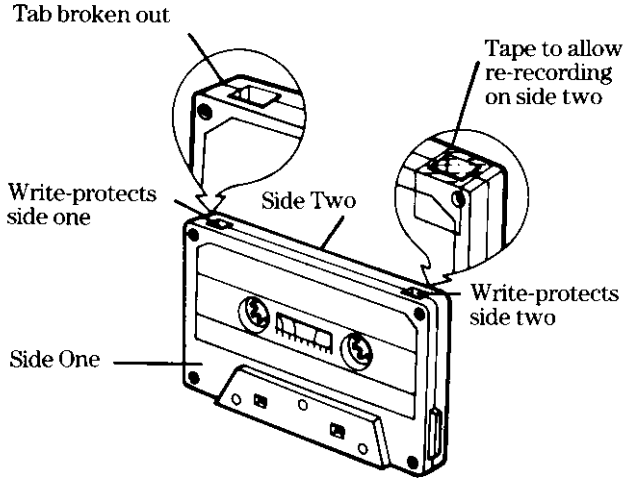
- Do not record on the leader (the non-magnetic segment at the beginning and at the end of a cassette tape).
- Do not record too close to the end of the tape. If the tape runs out before recording is complete, the whole file is unusable because the end-of-file record is needed for the TI-74 to access the file.
- Leave a few seconds of space after a file before you begin recording a new file. If you record in the middle of an existing file, the old file is no longer usable.
- Record and play back files on the same cassette recorder. Because of calibration differences between recorders, data recorded by one recorder may not read reliably on another recorder.

Backing Up Your Tapes

You may want to make a backup tape of your files. A backup tape provides another recording of your file in case one is accidentally lost or damaged. Because duplicating a tape may affect the accuracy of the recording, be sure to make two original recordings on two different tapes.

Preventing Erasure

Each time you make a recording, any material that you previously recorded on that portion of the tape is automatically erased. If you have a recording that you wish to keep permanently, break the rear left tab of the side you want to save. This prevents you from depressing the RECORD button on the recorder.



Note: If you need to record on a tape that has a tab missing, place a piece of cellophane tape over the tab opening before you record a file.

Procedure for Saving Programs

You can use a cassette recorder to record any programs you develop with the TI-74. Later, you can retrieve the programs by loading the information back into the TI-74's memory. Be sure to make the test recording given in "Determining the Recorder Settings" to find the correct tone and volume settings before you begin saving files.

Using the SAVE Command

You can use the SAVE command to copy (or save) a program in the TI-74's memory to a cassette tape.

To save the program currently in memory:

1. Type **SAVE "1.*file-name*"** and press **[ENTER]**.

   The 1 is the device number of the cassette recorder and *file-name* is the name of your file.

2. When the prompt `Position tape; then press ENTER` appears, advance the tape to a blank area with sufficient space to record the program.

   Be sure to leave some space between the last file and the beginning of the new file.

   If your recorder is equipped with a digital counter, rewind the tape to the beginning and reset the tape counter to zero before positioning the tape.

3. Press the **[ENTER]** key.

4. When the prompt `Press RECORD; then ENTER` appears, press the RECORD button on your recorder, and then press the **[ENTER]** key.

   After a few seconds, the prompt `Writing...` appears. The TI-74 is transferring information to the cassette file.

   Note: If you need to cancel the output operation, press the **[RESET]** key on the TI-74.

5. When the prompt `Press STOP; then ENTER` appears, press the STOP button on your recorder and then press **[ENTER]**.

The program is now saved on cassette tape. You may want to verify that the recording is accurate by using the VERIFY command.

Verifying Program Storage and Retrieval

The VERIFY command compares the program file to the program in memory. Using the VERIFY command is a good practice after saving or loading a program.

Using the VERIFY Command

To verify that a file was stored correctly or that it was loaded into memory correctly, follow these steps.

1. Type **VERIFY "1.*file-name*"** and press **[ENTER]**.

2. When the prompt `Position tape; then press ENTER` appears, perform one of the following.

   - If you know where your file is located on the cassette tape, press the FAST FORWARD and REWIND buttons to locate the beginning of the file.
   - If you are unsure where the file starts on the cassette tape, rewind the tape to the beginning.

3. Press **[ENTER]**.

4. When the prompt `Press PLAY; then ENTER` appears, press the PLAY button on your recorder. Then press **[ENTER]**.

   When the file is found, the prompt `Reading...` appears. The TI-74 is comparing the file on the cassette tape with the information in memory.

5. If the prompt `Press STOP; then ENTER` appears, the contents are the same. Press the STOP button on your recorder and then press the **[ENTER]** key.

   If an error message appears, the contents are not the same. The action you take at this point depends on whether you saved a program or retrieved a program.

   - If you saved a program, check the settings on the recorder, and then save the program again, using the procedure on the previous page.
   - If you retrieved a program, check the settings on the recorder, and then try retrieving the program again, using the procedure on the following page.

Setting Up a Sample Program and Data File

The BASIC statements that can involve cassette operations are SAVE, OLD, VERIFY, OPEN, CLOSE, INPUT, LINPUT, PRINT, EOF, and CALL I/O. A sample program using many of these statements is given below. A description of each statement can be found in Chapter 2.

The Sample Program

With the following program, you can enter information about three different accounts and retrieve the information as needed.

Step 1: Entering the Program

Enter the sample program listed below. Notice that line 210 includes commas to separate the data fields of the record.

```
100 REM Sample program
110 DISPLAY"1 – create; 2 – recall; 3 – quit"
120 PAUSE 2
130 DISPLAY "YOUR CHOICE? ";
140 ACCEPT AT(14) VALIDATE("123"),Z
150 ON Z GOTO 160,250,340
160 FOR X=1 TO 3 !key in data
170 INPUT "ACCT # ";A$(X),"ACCT NAME ";B$(X),
    "ACCT BALANCE ";A(X)
180 NEXT X
190 OPEN #1,"1.DATA1",DISPLAY,OUTPUT !create data file
200 FOR X=1 TO 3
210 PRINT #1,A$(X);",";B$(X);",";A(X)
220 NEXT X
230 CLOSE #1
240 GOTO 100 !return to menu
250 OPEN #1,"1.DATA1",DISPLAY,INPUT !read data file
260 FOR X=1 TO 3
270 INPUT #1,A$(X),B$(X),A(X)
280 NEXT X
290 CLOSE #1
300 FOR X=1 TO 3 !display data read from file
310 PRINT "ACCT # ";A$(X);" ";B$(X);" ";A(X):PAUSE
320 NEXT X
330 GOTO 100 !return to menu
340 END
```

You may want to run the program before storing it on cassette tape, or you may want to store the program at this point. (Refer to "Procedure for Saving Programs" for instructions.)

Step 2: Creating and Saving a Data File

1. Run the sample program.

   The following prompt appears in the display.

   ```
   1-create; 2-recall; 3-quit
   ```

   After two seconds, the following prompt appears in the display.

   ```
   YOUR CHOICE?
   ```

2. Enter 1 to create a data file. Display prompts instruct you to type in the account number, name, and balance for each account.

3. Enter the following information into the program.

   ```
   001
   Lang Institute
   30000

   002
   A. T. Optical
   50000

   003
   Grantham Const.
   75000
   ```

4. Follow the prompts to record the data file on cassette tape. (Refer to "Procedure for Saving Programs" for instructions.) Then you can select

   - Option 2 to start the prompts for retrieving the saved data file. (Refer to "Procedure for Retrieving Programs" for instructions.)
   - Option 3 to exit from the program.

If You Have Recording Difficulties

If you experience difficulty with your cassette recorder, you may be able to correct the problem by following these suggestions.

If the Recorder Is Not Operating

If your cassette recorder does not respond to any functions that move the tape, check the following.

- If the recorder is AC powered, check that it is connected to a functional AC power source.
- If the recorder is battery powered, check that its batteries are not depleted.
- If the interface is plugged into the remote control jack, unplug the remote plug and try storing or retrieving a file while operating the recorder manually. Be prepared to press a recorder button and **[ENTER]** at the same time when the computer prompts you.
- If the interface is plugged into the peripheral port on the PC-324 printer, disconnect the printer and connect the interface directly to the TI-74. If you can then store and retrieve a file, the printer may not be operating properly.

If You Cannot Locate a File

If you cannot locate a file at its tape counter setting, check the following.

- Be sure the volume is set correctly.
- Rewind the tape, reset the counter, and advance the tape to just before the noted tape counter position.

The default record length for a cassette recorder is 256 bytes.

Controlling the Printer From BASIC

This section contains information that applies when the PC-324 printer is connected to the TI-74. You can select options for the carriage return and for the line spacing. You can also send the printer instructions in the form of an IO subprogram or as character strings.

Device Code

For BASIC to address the printer, it must use the printer's device code, 12.

Carriage Return Options

After you use the OPEN statement to create a link to the printer, you can use the PRINT statement to print an item. When the item has been printed, the carriage return options cause either of two actions to occur.

- A carriage return can automatically follow the item. This option is selected by including R = L in the open statement or by not specifying R.
- The next print item can begin on the same line. This option is selected by including R = N in the OPEN statement.

These options can be used with the OPEN statement or the LIST command.

Example

When the following program is run, the resulting printout is as shown.

```
100 !R = N Option
110 OPEN  #1,"12.R = N",OUTPUT
120 PRINT  #1,"one  ":PRINT  #1,"two  ":PRINT  #1,"three  "

one  two  three
```

When the following program is run, the resulting printout is as shown.

```
100 !R = L Option
110 OPEN  #1,"12.R = L",OUTPUT
120 PRINT  #1,"one  ":PRINT  #1,"two  ":PRINT  #1,"three  "

one
two
three
```

Line Spacing Options

The line spacing options cause either single spacing or double spacing to occur.

- Printing can occur on every line of the paper. This option is selected by including L=S in the OPEN statement, or by not specifying L.
- Printing can occur on every other line of the paper. This option is selected by including L=D in the OPEN statement.

These options can be used with the OPEN statement or the LIST command.

Example

When the following program is run, the resulting printout is as shown.

```
100 !L=S Option
110 OPEN #1,"12.L=S",OUTPUT
120 PRINT #1,"one ":PRINT #1,"two ":PRINT #1,"three "

one
two
three
```

When the following program is run, the resulting printout is as shown.

```
100 !L=D Option
110 OPEN #1,"12.L=D",OUTPUT
120 PRINT #1,"one ":PRINT #1,"two ":PRINT #1,"three "

one

two

three
```


I/O Commands

The PC-324 supports certain peripheral commands available with the IO subprogram.

Code	Command	Result
80	Self Test #0	Reports the version of code in the printer's ROM. When CALL IO(12,80,X) is executed, X becomes 100 for version 1, 101 for version 2, etc.
81	Self Test #1	Performs a printing demonstration. When CALL IO(12,81,X) is executed, X becomes 6 if a device error occurs and zero if the demonstration runs to normal completion.
82	Self Test #2	Performs the printer's comparison with expected values. This test is similar to the calculator's power-up routine that checks for changes in memory contents. However, the values checked by this test are in ROM. Therefore, this test fails only if the printer is damaged. When CALL IO(12,82,X) is executed, X becomes 0 if the test is satisfactory and 80 if the test fails.

Example

The following program tests the printer.

```
110 X=0:Y=0
120 CALL IO(12,80,X)
130 CALL IO(12,82,Y)
140 X=X-99
150 PRINT USING"VERSION # IS IN THIS PRINTER";X:PAUSE
160 IF Y=0 THEN 170 ELSE 180
170 PRINT "PRINTER ROM IS SATISFACTORY":PAUSE:STOP
180 PRINT "PRINTER IS DAMAGED":PAUSE:STOP
```

Controlling the Printer From BASIC (Continued)

Printer Control Codes

You can signal the printer to perform any of the following actions by sending the corresponding ASCII code.

Code	Action
13	Carriage return
17	Use single spacing
18	Use double spacing

Because the carriage return includes a line feed automatically, the ASCII code for line feed (10) is ignored.

Example

The following program allows you to type lines and send them to the printer.

```
110 ACCEPT ERASE ALL,A$
120 DISPLAY "Double spacing?(Y/N)"
130 B$=KEY$
140 IF B$="Y" OR B$="y" THEN C=18 ELSE C=17
150 OPEN #1,"12",OUTPUT:PRINT #1,CHR$(C);A$
160 CLOSE #1:GOTO 110
```

Other ASCII Codes

The PC-324 responds to ASCII as listed in Appendix B except for the codes 00 through 31, the codes past 127, and four of the characters. The printer regards the codes 00 through 31 as NOP except the three listed above. The highest code to which the printer responds is decimal 127.

The list below shows the four codes whose characters differ from those of the TI-74.

Code	Character
92	\
124	¦

Code	Character
126	~
127	Space

Accessing Cartridge Memory

You can use the 8K Constant Memory cartridge for either of two purposes: to increase the memory available for BASIC programming or to store the contents of memory for later retrieval.

Adding the Cartridge Memory to BASIC

After the 8K Memory cartridge is installed in the cartridge port, a CALL ADDMEM command must be executed if you intend to use cartridge memory as part of available memory.

Caution: Executing a CALL ADDMEM command erases the contents of cartridge memory before adding the cartridge memory to BASIC. However, the computer's memory is unaffected.

Refer to Chapter 2 for information about the ADDMEM subprogram and the FRE function.

Cancelling Memory Expansion

After CALL ADDMEM has been executed, the cartridge memory remains available to BASIC until one of the following occurs.

- A NEW ALL command is executed.
- The **[RESET]** key is pressed.
- The cartridge is removed while the calculator is on.
- Battery power is lost.

When one of these conditions occurs, the cartridge memory is separated from the available BASIC memory and the computer's memory is cleared.

Using a Cartridge For Storage

The 8K Memory cartridge can be used to store the entire contents of the TI-74's 8K memory, known as a memory image. Refer to Chapter 2 for information about the GET and PUT subprograms, which handle the transfer of information to a cartridge.

The cartridge retains a memory image until:

- Another image is placed in the cartridge.
- CALL ADDMEM is executed.
- Battery power is lost.

Note: If the cartridge is in use as memory expansion, you must cancel memory expansion before you can use the cartridge for storage.

Appendices

The appendices provide information you may need when investigating the details for a program. Note that a difficulty section is included in the TI-74 *User's Guide* that can help correct certain problems. That book also has information about contacting Texas Instruments and contains the warranty.

Table of Contents

Appendix A: Reserved Word List

The following is a list of all TI-74 BASIC reserved words. A reserved word may not be used as a variable name, but may be a portion of a variable name. A variablename can also be a portion of a reserved word except for the abbreviations of reserved words.

Commands and Statements

Most statements can be executed immediately as well as used in a program line. The words you can include only in a program statement are noted with a [P] and the commands you can use only outside the program are noted with a [C].

ABS
ACCEPT[P]
ACOS
ACOSH
ALL
ALPHA
ALPHANUM
AND
APPEND
ASC
ASIN
ASINH
AT
ATANH
ATN
BREAK
CALL
CHR$
CLOSE
CON[C]
CONTINUE[C]
COS
COSH
DATA
DEG
DEL
DELETE
DIGIT
DIM
DISPLAY
ELSE
END
EOF
ERASE
ERROR
EXP
FOR
FORMAT
FRE
GOSUB[P]
GOTO[P]
GRAD
IF
IMAGE[P]
INPUT[P]
INT
INTERNAL
KEY$
LEN
LET
LINPUT[P]
LIST[C]
LN
LOG
NEW[C]
NEXT
NOT
NULL
NUM[C]
NUMBER[C]
NUMERIC
OLD[C]
ON
OPEN
OR
OUTPUT

Commands and Statements (Continued)

PAUSE
PI
POS
PRINT
PROTECTED
RAD
RANDOMIZE
READ[P]
REC
RELATIVE
REM
REN[C]
RENUMBER[C]
RESTORE
RETURN[P]
RND
RPT$
RUN
SAVE[C]
SEG$
SGN
SIN
SINH
SIZE
SQR
STEP
STOP
STR$
SUB[P]
SUBEND[P]
SUBEXIT[P]
TAB
TAN
TANH
THEN
TO
UALPHA
UALPHANUM
UNBREAK
UPDATE
USING
VAL
VALIDATE
VARIABLE
VERIFY[C]
WARNING
XOR

Subprograms

You can use the names of system subprograms as variable names. However, subprograms you write should not be given system subprogram names.

CALL ADDMEM[C]
CALL ERR
CALL GET[C]
CALL IO
CALL KEY
CALL PUT[C]

Appendix B: ASCII Character Codes

The following table lists the ASCII character codes in decimal and hexadecimal notation.

ASCII Table

The ASCII commands and/or character(s) displayed when the key or key sequence is pressed are shown in the column titled **Character**.

System-reserved character codes (0–15) and the user-assigned keys (codes 128–137) are shown as two asterisks (**).

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
00	00	NULL	**	[CTL] 0
01	01	SOH	**	[CTL] A
02	02	STX	**	[CTL] B
03	03	ETX	**	[CTL] C
04	04	EOT	**	[CTL] D
05	05	ENQ	**	[CTL] E
06	06	ACK	**	[CTL] F
07	07	BEL	**	[CTL] G
08	08	BS	**	[CTL] H
09	09	HT	**	[CTL] I
10	0A	LF	**	[CTL] J
11	0B	VT	**	[CTL] K
12	0C	FF	**	[CTL] L
13	0D	CR	**	[CTL] M or [ENTER]
14	0E	SO	**	[CTL] N
15	0F	SI	**	[CTL] O
16	10	DLE		[CTL] P
17	11	DC1		[CTL] Q
18	12	DC2		[CTL] R
19	13	DC3		[CTL] S
20	14	DC4		[CTL] T
21	15	NAK		[CTL] U
22	16	SYN		[CTL] V
23	17	ETB		[CTL] W
24	18	CAN		[CTL] X
25	19	EM		[CTL] Y
26	1A	SUB		[CTL] Z
27	1B	ESC		[CTL] [CLR] or [FN] [SPACE]
28	1C	FS		[CTL] [+/–]

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
29	1D	GS		[CTL] ;
30	1E	RS		[CTL] .
31	1F	US		[CTL] ,
32	20	SPACE	SPACE	[SPACE]
33	21	!	!	[SHIFT] 1
34	22	"	"	[SHIFT] 2
35	23	#	#	[SHIFT] 3
36	24	$	$	[SHIFT] 4
37	25	%	%	[SHIFT] ,
38	26	&	&	[SHIFT] 5
39	27	'	'	[SHIFT] [SPACE]
40	28	(	(	[SHIFT] [↑]
41	29	)	)	[SHIFT] [↓]
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	[SHIFT] ;
59	3B	;	;	;
60	3C	<	<	[SHIFT] 0
61	3D	=	=	[SHIFT] [ENTER]
62	3E	>	>	[SHIFT] .
63	3F	?	?	[SHIFT] [+/–]
64	40	@	@	[CTL] 2
65	41	A	A	[SHIFT] A
66	42	B	B	[SHIFT] B

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
67	43	C	C	[SHIFT] C
68	44	D	D	[SHIFT] D
69	45	E	E	[SHIFT] E
70	46	F	F	[SHIFT] F
71	47	G	G	[SHIFT] G
72	48	H	H	[SHIFT] H
73	49	I	I	[SHIFT] I
74	4A	J	J	[SHIFT] J
75	4B	K	K	[SHIFT] K
76	4C	L	L	[SHIFT] L
77	4D	M	M	[SHIFT] M
78	4E	N	N	[SHIFT] N
79	4F	O	O	[SHIFT] O
80	50	P	P	[SHIFT] P
81	51	Q	Q	[SHIFT] Q
82	52	R	R	[SHIFT] R
83	53	S	S	[SHIFT] S
84	54	T	T	[SHIFT] T
85	55	U	U	[SHIFT] U
86	56	V	V	[SHIFT] V
87	57	W	W	[SHIFT] W
88	58	X	X	[SHIFT] X
89	59	Y	Y	[SHIFT] Y
90	5A	Z	Z	[SHIFT] Z
91	5B	[	[	[CTL] 8
92	5C	¥	¥	[CTL] /
93	5D	]	]	[CTL] 9
94	5E	^	^	[SHIFT] 6
95	5F	_	_	[CTL] 5
96	60	`	`	[CTL] 3
97	61	a	a	A
98	62	b	b	B
99	63	c	c	C
100	64	d	d	D
101	65	e	e	E
102	66	f	f	F
103	67	g	g	G
104	68	h	h	H

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
105	69	i	i	I
106	6A	j	j	J
107	6B	k	k	K
108	6C	l	l	L
109	6D	m	m	M
110	6E	n	n	N
111	6F	o	o	O
112	70	p	p	P
113	71	q	q	Q
114	72	r	r	R
115	73	s	s	S
116	74	t	t	T
117	75	u	u	U
118	76	v	v	V
119	77	w	w	W
120	78	x	x	X
121	79	y	y	Y
122	7A	z	z	Z
123	7B	{	{	[CTL] 6
124	7C	|	|	[CTL] 1
125	7D	}	}	[CTL] 7
126	7E	→	→	[CTL] 4
127	7F		←	[SHIFT] [→]
128	80	**		[FN] 0
129	81	**		[FN] 1
130	82	**		[FN] 2
131	83	**		[FN] 3
132	84	**		[FN] 4
133	85	**		[FN] 5
134	86	**		[FN] 6
135	87	**		[FN] 7
136	88	**		[FN] 8
137	89	**		[FN] 9
138	8A			
139	8B			
140	8C			
141	8D			[SHIFT] /
142	8E			[SHIFT] *

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
143	8F			[SHIFT] –
144	90			[SHIFT] +
145	91			[CTL] *
146	92			[CTL] –
147	93			[CTL] +
148	94	NEW		[FN] [←]
149	95	NUMBER		[FN] [→]
150	96	RENUMBER		[FN] [↑]
151	97	FRE(		[FN] [↓]
152	98	VERIFY		[FN] /
153	99	SAVE		[FN] *
154	9A	OLD		[FN] –
155	9B	LIST		[FN] +
156	9C	FORMAT		[FN] .
157	9D	OPEN		[FN] ,
158	9E	ERROR		[FN] ;
159	9F	SGN(		[FN] [+/–]
160	A0	DELETE		[FN] [CLR]
161	A1	FOR	｡	[FN] A
162	A2	READ	｢	[FN] B
163	A3	PAUSE	｣	[FN] C
164	A4	NEXT	､	[FN] D
165	A5	TAN(	･	[FN] E
166	A6	IF	ｦ	[FN] F
167	A7	THEN	ｧ	[FN] G
168	A8	ELSE	ｨ	[FN] H
169	A9	SQR(	ｩ	[FN] I
170	AA	GOTO	ｪ	[FN] J
171	AB	GOSUB	ｫ	[FN] K
172	AC	RETURN	ｬ	[FN] L
173	AD	RESTORE	ｭ	[FN] M
174	AE	DATA	ｮ	[FN] N
175	AF	CHR$(	ｯ	[FN] O
176	B0	CALL	ｰ	[FN] P
177	B1	SIN(	ｱ	[FN] Q
178	B2	PI	ｲ	[FN] R
179	B3	TO	ｳ	[FN] S
180	B4	LN(	ｴ	[FN] T

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
181	B5	EXP(	ｵ	[FN] U
182	B6	INPUT	ｶ	[FN] V
183	B7	COS(	ｷ	[FN] W
184	B8	USING	ｸ	[FN] X
185	B9	LOG(	ｹ	[FN] Y
186	BA	PRINT	ｺ	[FN] Z
187	BB	BREAK	ｻ	[FN] [BREAK]
188	BC	STOP	ｼ	[SHIFT] [RUN]
189	BD		ｽ	
190	BE	CONTINUE	ｾ	[FN] [RUN]
191	BF	RUN	ｿ	[RUN]
192	C0		ﾀ	
193	C1		ﾁ	
194	C2		ﾂ	
195	C3		ﾃ	
196	C4		ﾄ	
197	C5		ﾅ	
198	C6		ﾆ	[+/–]
199	C7		ﾇ	
200	C8		ﾈ	
201	C9		ﾉ	
202	CA		ﾊ	
203	CB		ﾋ	
204	CC		ﾌ	
205	CD		ﾍ	
206	CE		ﾎ	
207	CF		ﾏ	
208	D0		ﾐ	[SHIFT] [FN] 0
209	D1		ﾑ	[SHIFT] [FN] 1
210	D2		ﾒ	[SHIFT] [FN] 2
211	D3		ﾓ	[SHIFT] [FN] 3
212	D4		ﾔ	[SHIFT] [FN] 4
213	D5		ﾕ	[SHIFT] [FN] 5
214	D6		ﾖ	[SHIFT] [FN] 6
215	D7		ﾗ	[SHIFT] [FN] 7
216	D8		ﾘ	[SHIFT] [FN] 8
217	D9		ﾙ	[SHIFT] [FN] 9
218	DA		ﾚ	

ASCII Table (Continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
219	DB		□	
220	DC		▗	
221	DD		▐	
222	DE		▝	
223	DF		▘	
224	E0		α	
225	E1		ß	[SHIFT] [←]
226	E2		Γ	
227	E3		π	
228	E4		Σ	
229	E5	PB	σ	[SHIFT] 9
230	E6	OFF	µ	[OFF]
231	E7	BREAK	τ	[BREAK]
232	E8	UP	Φ	[↑]
233	E9	DOWN	Θ	[↓]
234	EA		Ω	
235	EB		δ	
236	EC		∞	
237	ED		∅	
238	EE		∈	
239	EF		∩	
240	F0	MODE	≡	[MODE]
241	F1		±	
242	F2		≥	
243	F3		≤	
244	F4		⌠	
245	F5		⌡	[CTL] [RUN]
246	F6	DEL	÷	[SHIFT] 7
247	F7	INS	≈	[SHIFT] 8
248	F8	HOME	°	[CTL] [↑]
249	F9	DEL REST	∙	[CTL] [↓]
250	FA	CLR	√	[CLR]
251	FB	BTAB	ⁿ	[CTL] [←]
252	FC	LEFT	²	[←]
253	FD	FTAB	■	[CTL] [→]
254	FE	RIGHT		[→]
255	FF		■	[CTL] [SPACE]

The logical operators AND, OR, NOT, and XOR can be used on integer numbers in the range −32768 to 32767. This appendix briefly describes the binary number system, conversion of decimal numbers to their binary equivalents, and the operation of the logical operators.

Number Systems

Binary (base 2) notation is another way to express the value of a number. Our usual system, decimal (base 10) notation, uses combinations of the ten digits zero through nine. Numbers written in binary notation use only the two digits zero and one. Each position occupied by a binary digit (a 0 or 1) is called a bit.

Decimal Notation

In decimal notation, each digit in a number represents a power of 10. For example, the number 2408 in decimal notation can be written in expanded form as follows.

$$(2\times10^3)+(4\times10^2)+(0\times10^1)+(8\times10^0)$$

This is equal to 2408 as shown below.

$$\begin{aligned} 2\times10^3 &= 2\times1000 = 2000\\ 4\times10^2 &= 4\times\ 100 = \ 400\\ 0\times10^1 &= 0\times\ 10 = \ \ 0\\ 8\times10^0 &= 8\times\ \ 1 = \ \ 8\\ &\qquad\qquad\quad\ 2408 \end{aligned}$$

Binary Notation

In binary notation, each digit represents a power of two. For example, the binary number 101101 can be written as

$$(1\times2^5)+(0\times2^4)+(1\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)$$

For reference purposes, the powers of two and their decimal values are as follows.

2^{14}	2^{13}	2^{12}	. . .	2^3	2^2	2^1	2^0
16384	8192	4096	. . .	8	4	2	1

For the TI-74, the 2^{15} bit must be zero to interpret the bits as stated here.

Binary Notation (Continued)

The decimal equivalent of 101101 can be calculated as shown below.

$$\begin{aligned} 1\times2^5 &= 1\times32 = 32\\ 0\times2^4 &= 0\times16 = 0\\ 1\times2^3 &= 1\times8 = 8\\ 1\times2^2 &= 1\times4 = 4\\ 0\times2^1 &= 0\times2 = 0\\ 1\times2^0 &= 1\times1 = \underline{1}\\ &\qquad\qquad 45 \end{aligned}$$

To convert a number from decimal notation to binary notation, repeatedly reduce the decimal number by the greatest power of 2 not larger than the number until there is no remainder.

Example

The decimal number 77 can be converted to binary notation using the following technique.

The largest power of 2 contained in the number 77 is 64 (2^6). A 1 is placed in that position of the binary number as shown below.

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	0

Reducing 77 by 64 leaves a remainder of 13. The largest power of 2 contained in 13 is 8 (2^3) and a 1 is placed there. Reducing 13 by 8 leaves a remainder of 5. The largest power of 2 contained in 5 is 4 (2^2) and a 1 is placed there. Reducing 5 by 4 leaves a remainder of 1. Place a 1 in the 2^0 position.

The decimal number 77 in binary notation is shown below.

128	64	32	16	8	4	2	1
0	1	0	0	1	1	0	1

Example (Continued)

You can check the accuracy of the conversion as follows.

$$\begin{aligned} 1\times2^6 &= 1\times64 = 64\\ 0\times2^5 &= 0\times32 = 0\\ 0\times2^4 &= 0\times16 = 0\\ 1\times2^3 &= 1\times8 = 8\\ 1\times2^2 &= 1\times4 = 4\\ 0\times2^1 &= 0\times2 = 0\\ 1\times2^0 &= 1\times1 = \underline{1}\\ &\qquad\qquad 77 \end{aligned}$$

Logical Operations

When logical operations are performed on numbers within the valid range, the TI-74 first converts the values to their 16-bit binary equivalents. The logical operations are performed on a bit-by-bit basis, and the resulting binary number is converted back to decimal notation.

The left-most bit is reserved to indicate the sign (0 = positive; 1 = negative). Therefore, the largest number that can be represented by the remaining 15 bits is 32,767.

If a decimal number with a fractional part is used with a logical operator, the number is rounded before any logical operation is performed.

The following are the rules for the four logical operators.

Operator	Rule
AND	If both bits are 1s, the result is 1. If either bit is 0, the result is 0.
OR	If either bit is a 1, the result is 1. If both bits are 0, the result is 0.
XOR	If either bit, but not both, is 1, the result is 1. If both bits are the same, the result is 0.
NOT	If the bit is 0, the result is 1. If the bit is 1, the result is 0.

Example

When the logical operations are performed on the numbers 77 and 67, the numbers are first converted to binary notation. The number 77 is represented in 16 bits as 0000000001001101 and the number 67 is represented in 16 bits as 0000000001000011. The results of performing an AND, an OR, and an XOR on the two values are shown below.

```
        AND
(77)    0000000001001101
(67)    0000000001000011
(65)    0000000001000001

        OR
(77)    0000000001001101
(67)    0000000001000011
(79)    0000000001001111

        XOR
(77)    0000000001001101
(67)    0000000001000011
(14)    0000000000001110
```

The results of performing an AND, OR, and an XOR on 77 and 67 can be obtained on your TI-74 by entering the following.

```
PRINT 77 AND 67; 77 OR 67; 77 XOR 67
```


Negative Binary Numbers

Using the operator NOT on 77 and 67 is shown below.

```
        NOT 77                          NOT 67
(77)    0000000001001101        (67)    0000000001000011
(-78)   1111111110110010        (-68)   1111111110111100
```

To display the results of NOT 77 and NOT 67, enter:

```
PRINT NOT 77; NOT 67
```

Note that the results of NOT 77 and NOT 67 have a 1 in the left-most bit that denotes that they represent negative numbers. In the TI-74 a negative binary number is represented as the two's complement of the absolute value of the number.

To obtain the two's complement of a binary number, change each 0 bit to 1 and each 1 bit to 0. Then add 1 to this changed number. For example, the two's complement of 77 is obtained as shown below.

```
77 in binary        0000000001001101
Change each bit     1111111110110010
Add 1                              1
-77 in binary       1111111110110011
```

A more detailed description of binary arithmetic is beyond the scope of this appendix. Refer to a standard reference book on this subject for more information.

Hierarchy of Operators

The numeric operators on the TI-74 have a designated priority of completion as follows. However, you can use parentheses to group operations in any order regardless of the hierarchy.

Unary operators (including NOT)
Arithmetic Operators
Relational Operators
Logical Operators (XOR, AND, OR)

Appendix D: Error Messages

The following lists describe each error message generated by the TI-74. The first list, arranged alphabetically by message, provides detailed information about probable error causes. The second list, arranged in ascending order by error code, serves as a cross reference to locate the message associated with a particular error code.

Coping With Errors

When an error message is displayed, the [→], [←], [↑], [↓] and **[SHIFT] [PB]** keys can be used to display additional system error information and to edit an erroneous line.

[SHIFT] [PB] is used when an error occurs after a line is entered. **[SHIFT] [PB]** displays the erroneous entry that can then be edited and entered again.

[↑], [↓] used when an error occurs during program execution to display the program line that was executing when the error occurred.

Errors can be handled in a program using ON ERROR and CALL ERR. Refer to chapter 2 for more information.

Messages Listed Alphabetically

`Bad argument`

- Invalid argument provided for one of the built-in numeric, string, or file functions such as LOG, CHR$, or EOF.
- Invalid argument provided for one of the option clauses in an input/output statement such as AT, SIZE, VALIDATE, and TAB.
- Arguments in a CALL statement did not match the requirements for the subprogram called.

`Bad data`

- Entered more than one value at a time in an INPUT or ACCEPT statement.
- Invalid data from a file in an INPUT or LINPUT statement.

Messages Listed Alphabetically (Continued)

`Bad dimension`

- Specified array dimension was negative or was not a numeric constant.
- Too many elements specified for an array.
- More than three dimensions specified for an array.
- Missing comma between dimensions or missing parentheses around dimensions of an array.
- Subscript value too large.
- Missing comma between subscripts or missing parentheses around subscripts.
- Incorrect number of subscripts.

`Bad program type`

- Entered a BASIC program line with a Pascal or other non-BASIC program in memory.
- Entered a SAVE, VERIFY, BREAK *line-list*, UNBREAK *line-list*, NUMBER, RENUMBER, LIST, CONTINUE *line-number*, RUN *line-number*, or DELETE *line-group* command with a Pascal or other non-BASIC program in memory.
- Attempted to CALL a main program or RUN a subprogram.

`Bad value`

- Index value in ON GOTO or ON GOSUB statement was zero or greater than the number of line number entries.
- Raised a negative value to a non-integer power.
- Invalid value provided for one of the option clauses in an input/output statement such as AT, SIZE, REC, and VARIABLE.
- Attempted a logical operation (AND, OR, XOR, or NOT) with a value less than −32768 or greater than 32767.

Messages Listed Alphabetically (Continued)

`Break`
- A breakpoint occurred or the break key was pressed.

`Can't do that`
- Attempted to perform a string operation as an immediate calculation.
- Entered CONTINUE command when not stopped at a breakpoint.
- A SUBEXIT or SUBEND statement was encountered when no subprogram was called. For example, CONTINUE *line-number* specified a line in a subprogram after the main program stopped at a breakpoint.

`Complex`
- Too many functions, operators, or levels of parentheses pending evaluation; expression must be simplified or performed in two or more steps in separate statements.

`contents may be lost`
- When the power was turned on, the computer determined that the contents of memory are not the same as when the power was turned off. However, some system data was correct, so the loss may or may not be serious. This message often appears when the **[RESET]** key is pressed while the power is on.

`DATA error`
- Out of data in the current program or subprogram.
- Improper data list in a DATA statement. For example, items not separated by commas.
- During an attempt to read a numeric item, the data read was not a valid representation of a numeric constant.

Messages Listed Alphabetically (Continued)

`Division by zero`
- Evaluation of a numeric expression includes division by zero; result is replaced by 9.9999999999999E+127 with the appropriate algebraic sign.

`Extension`
- Attempted to execute an extended BASIC statement or function without the extension in the system.
- May also occur when the contents of memory have been improperly modified (see `System error`).

`File error`
- *File-number* specified in an OPEN statement refers to a file already opened.
- *File-number* in an input/output statement, other than OPEN, did not refer to an open file.
- *File-number* or *device-number* in an input/output statement was greater than 255.
- Attempted to INPUT or LINPUT from a file opened in OUTPUT or APPEND mode.
- Attempted to LINPUT from an internal-type file.
- Attempted to PRINT to a file opened in INPUT mode.
- Used REC clause in an input/output statement that accessed a sequential file.
- Missing period or comma after device number in *device* or *filename* specification.

Messages Listed Alphabetically (Continued)

`FOR/NEXT error`

- More FOR statements than NEXT statements in a program or subprogram. **Note:** The line number reported is the last line of the current program or subprogram, not the line containing the unmatched FOR statement.
- More NEXT statements than FOR statements in a program or subprogram.
- *Control-variable* in NEXT statement did not match *control-variable* in corresponding FOR statement.
- Executed a NEXT statement without previously executing the corresponding FOR statement.
- Too many levels of nested FOR NEXT loops.
- Same control variable used in nested FOR NEXT loops.

`IMAGE error`

- Null string provided as image string.
- Numeric format field specified more than 14 significant digits.
- *Print-list* included a print-item but image string had only literal characters.

`In use`

- Called an active subprogram; subprograms may not call themselves, directly or indirectly.

`Initialized`

- Displayed when circumstances force the complete initialization of the system. The system is initialized when the power is turned on and the computer determines that:
  - the contents of memory have been destroyed (may occur after changing the batteries).
  - expansion RAM previously appended through the ADDMEM subprogram is no longer in the system.
- May also appear when the **[RESET]** key is pressed (much of the same memory checking is performed).

Messages Listed Alphabetically (Continued)

`I/O error`

- An error was returned by a peripheral device during an input/output (I/O) statement or command, or while using the EOF function. A special I/O code is returned by the device and is displayed after the message. Common I/O error codes are described in the I/O ERROR CODES section of this appendix. I/O error codes for the cassette recorder are listed later in this appendix.

  The error code is followed by the *file-number* or the *device-number*, whichever is appropriate to the statement or command being executed. A number sign indicates a *file-number* and quotation marks indicate a *device-number*. Both the common codes and other device-dependent I/O error codes are described in the peripheral manuals.

`Line number error`

- Could not find a line number specified in BREAK, CONTINUE, DELETE, GOSUB, GOTO, ON ERROR, USING, RESTORE, RUN, or BREAK.
- RENUMBER could not find a referenced line. The command replaced the reference by 32767, which is not a valid line number.
- BASIC statement referred to a line number that was lower than the first (or higher than the last) line number of the current program or subprogram.
- Line number specified in a statement or command was less than 1 or greater than 32766.
- RENUMBER command generated a line number greater than 32766.

Messages Listed Alphabetically (Continued)

`Memory full`
- Insufficient space to add, insert, or edit a program line.
- Insufficient space to allocate variables for a program or subprogram.
- Insufficient memory to allocate space for a string value.
- Insufficient space to load a program or subprogram into memory.
- Insufficient space to OPEN a file or device.
- Insufficient space to assign a user-assigned string.

`Mismatch`
- Used a string argument where a numeric argument was expected or a numeric argument where a string argument was expected.
- Assigned a string value to a numeric variable or a numeric value to string variable.
- A numeric variable or expression was provided as a prompt in an INPUT or LINPUT statement.

`Missing statement`
- An error-processing subroutine terminated with a SUBEXIT or SUBEND statement instead of a RETURN statement.
- SUBEND missing in a subprogram.
- Encountered a SUB statement within a subprogram; a subprogram cannot contain another subprogram.
- Executed a RETURN statement without previously executing the corresponding GOSUB statement.

Messages Listed Alphabetically (Continued)

`Name table full`
- Defined more than 95 variable names.

`No RAM`
- Called ADDMEM subprogram with no cartridge installed or with a cartridge that had no available RAM.
- Called PUT or GET with no cartridge installed or with a cartridge that has no RAM.

`Not defined`
- Attempted to perform a calculation with a variable that has not been defined.
- Encountered an undefined variable in a program or subprogram. This error can occur when CONTINUE *line-number* specifies a line that is not in the same program or subprogram where the breakpoint occurred.

`Not found`
- RUN statement did not find the specified program.
- CALL statement did not find the specified subprogram.

`Overflow`
- A numeric value was entered or a numeric expression was evaluated that resulted in a number whose absolute value was greater than 9.9999999999999E + 127; the value is replaced by 9.9999999999999E + 127 with the appropriate algebraic sign.

`Parenthesis`
- A statement or expression did not contain the same number of left and right parentheses.
- Left and right parentheses in a statement or expression did not match up. For example, SIN(1 +)PI/2)(where SIN(1 + (PI/2)) was intended.

Messages Listed Alphabetically (Continued)

`Previously defined`

- Variable in a DIM statement appeared previously in the current program or subprogram.
- Variable referenced using the wrong number of dimensions. For example, a variable was first used as a simple variable and later used as an array in the same program or subprogram.

`Protection error`

- Attempted to insert, delete, or edit a line with a protected program in memory.
- Attempted to LIST, SAVE, NUMBER, or RENUMBER a protected program.

`Stack underflow`

- Attempted to remove a value from the execution control stack when it was empty. This error only occurs when the contents of memory have been improperly modified (see `System error`).

`Syntax`

- Missing parentheses or quotation mark(s).
- Missing statement separator (:) or tail remark symbol (!).
- Missing or extra comma(s). For example:
  - between arguments in *argument-list*
  - between line numbers in *line-number-list*
  - between variables in *variable-list*
  - after *file-number* in input/output statements
- Missing hyphen in line sequence.
- Invalid character in statement. For example "%", "?", "~", "[", "]", etc., are valid only within quoted strings or in an IMAGE or REM statement.
- Invalid character within a numeric constant.

Messages Listed Alphabetically (Continued)

`Syntax` (Continued)

- Improperly placed keyword. For example:
  - DIM or SUBEND is used after a DIM statement in a multiple statement line
  - a statement begins with a non-statement keyword such as TO, ERROR, VARIABLE, SIZE
  - a misspelled variable results in a keyword or a misspelled keyword in a variable
  - a keyword is used as a variable, such as ON VAL GOTO or IF STOP = 1 THEN
- Missing keyword. For example:
  - no TO after FOR
  - no THEN after IF
  - no GOTO or GOSUB after ON *numeric-expression*
  - no STOP, NEXT, or ERROR after ON BREAK
  - no PRINT, NEXT, or ERROR after ON WARNING
- SUB statement used after the first statement in a multiple statement line.
- Statement other than REM, !, END, or SUB used after a SUBEND statement.
- Missing or invalid *filename* in OLD, SAVE, VERIFY, or DELETE file command.
- Duplicated option in input/output statement. For example:
  - more than one AT, SIZE, ERASE ALL is in ACCEPT or DISPLAY
  - more than one string expression is in VALIDATE
  - more than one *open-mode*, *file-type*, *file-organization* is in OPEN
- Missing argument or clause. For example:
  - no limit value after TO or increment value after STEP
  - no line number or statement after THEN or ELSE
  - no *string-constant* following IMAGE
  - no *line-number* or *string-expression* after USING
  - no value before or no value after a binary operator such as *, /, ^, or &
  - no input variable following INPUT, LINPUT, ACCEPT, or READ

Messages Listed Alphabetically (Continued)

Syntax (Continued)

- Invalid argument or clause. For example:
  - a string variable is used as *control-variable* in FOR
  - a numeric variable is used as input variable in LINPUT
  - VALIDATE or NULL is used in a DISPLAY statement
  - USING or TAB is used with an internal-type file
  - the size of print item exceeds record size for an internal-type file
- ACCEPT, CALL with BASIC-language subprograms, GOSUB, GOTO, INPUT, LINPUT, ON ERROR *line-number*, ON GOSUB, ON GOTO, READ, RESTORE *line-number*, SUB, SUBEXIT, and SUBEND statements can be executed only in a program.
- SUBEXIT or SUBEND statement encountered in a main program.
- Used CALL ADDMEM, CONTINUE, DELETE *line-group*, LIST, NEW, NUMBER, OLD, RENUMBER, SAVE, or VERIFY in a program.

System error

- This error generally occurs when the contents of memory have been lost or improperly modified. For example, memory may be modified by a loss of power.

Too long

- The internal representation of a program line or immediate statement(s) was too long.
- The LIST representation of a program line exceeded 80 characters.
- More than 15 characters in a variable or subprogram name.

Truncation

- String operation (concatenation or RPT$) resulted in a string with more than 255 characters; the extra characters are discarded.

Error Codes Listed in Ascending Order

Code	Message
E0 or W0	I/O error
E1 or W1	Syntax
E2	Complex
E3 or W3	Mismatch
E4 or W4	Bad value
E5	Stack underflow
E6	FOR/NEXT error
E7 or W7	Bad data
E8	Bad dimension
E9	Previously defined
E10	Can't do that
E11	Line number error
E12	Missing statement
E13	Not found
E14	Bad program type
E15	Protection error
E16	In use
E17 or W17	Not defined
E18	Image error
E19	File error
E20	Name table full
E21 or W21	Parenthesis
E22	Too long
E23 or W23	Bad argument
E24	Extension missing
W25	Overflow
W26	Division by zero
W27	contents may be lost
W28	Truncation
W29	Break
W30	Initialized
E31	No RAM
E32	DATA error
E126	System error
E127	Memory full

I/O Error Codes

The following list details the standard input/output (I/O) error codes. Some peripherals may have additional error codes; if so, they are explained in the peripheral manual.

I/O errors are displayed in one of the following forms.

- `I/O error ccc #fff`
- `I/O error ccc "ddd"`

where `ccc` is the I/O error code listed below or in the peripheral manual, `fff` is the file number assigned in an OPEN statement, and `ddd` is the device number associated with the peripheral device.

Code	Definition
1	DEVICE/FILE OPTIONS ERROR ► Incorrect or invalid option specified in "*device.filename*". ► *Filename* too long or missing in "*device.filename*".
2	ERROR IN ATTRIBUTES ► In an OPEN statement, incorrect attributes (*file-type, file-organization, open-mode, record-length*) were specified for an existing file.
3	FILE NOT FOUND ► The file specified in one of the following operations does not exist. – OPEN statement using the INPUT attribute – OLD "*device.filename*" – RUN "*device.filename*" – DELETE "*device.filename*"
4	DEVICE/FILE NOT OPEN ► Attempted to access a closed file with a INPUT, LINPUT, PRINT, or CLOSE operation. ► File specified in EOF function is closed.

I/O Error Codes (Continued)

Code	Definition
5	DEVICE/FILE ALREADY OPEN ► Attempted to OPEN or DELETE an open file. ► Attempted to FORMAT storage medium on a device that has a file open.
6	DEVICE ERROR ► A failure has occurred in the peripheral. This error can occur when directory information on a medium was lost, the peripheral detected a transmission error or a medium failure, etc.
7	END OF FILE ► Attempted to read past the end of the file.
8	DATA/FILE TOO LONG ► Attempted to output a record that was longer than the capacity of the device. ► A file exceeded the maximum file length for a device.
9	WRITE PROTECT ERROR ► Attempted to FORMAT a write-protected storage medium. ► Attempted to OPEN a write-protected file in OUTPUT or UPDATE mode. ► Attempted to DELETE a file from a write-protected medium.
10	NOT REQUESTING SERVICE ► Response to a service request poll when the specified device did not request service. (This code is used in special applications and should not be encountered during normal execution of BASIC programs.)
11	DIRECTORY FULL ► Attempted to OPEN a new file on a device whose directory is full.

I/O Error Codes (Continued)

Code	Definition
12	BUFFER SIZE ERROR ► When an existing file was opened for input or update, the specified record length (VARIABLE XXX) was less than the length of the largest record in the existing file. ► The VERIFY command found the program in memory was smaller than the program on the storage medium.
13	UNSUPPORTED COMMAND ► Attempted an operation not supported by the peripheral.
14	DEVICE/FILE NOT OPENED FOR OUTPUT ► Attempted to write to a file or device opened for input.
15	DEVICE/FILE NOT OPENED FOR INPUT ► Attempted to read from a file or device opened for output or append.
16	CHECKSUM ERROR ► The checksum calculated on the input record was incorrect.
17	RELATIVE FILES NOT SUPPORTED ► Device specified in OPEN does not support relative record file organization.
19	APPEND MODE NOT SUPPORTED ► Device specified in OPEN statement does not support append mode.
20	OUTPUT MODE NOT SUPPORTED ► Device specified in OPEN statement does not support output mode.

I/O Error Codes (Continued)

Code	Definition
21	INPUT MODE NOT SUPPORTED ► Device specified in OPEN statement does not support input mode.
22	UPDATE MODE NOT SUPPORTED ► Device specified in OPEN statement does not support update mode.
23	FILE TYPE ERROR ► File type specified in OPEN statement is not supported by the specified device. ► File type specified in OPEN statement does not match file type of existing file or device.
24	VERIFY ERROR ► Program or data in memory does not match specified program or storage medium.
25	LOW BATTERIES IN PERIPHERAL ► Attempted an I/O operation with a device whose batteries are low.
26	UNINITIALIZED MEDIUM ► Attempted to open a file on uninitialized storage medium. ► Attempted to open a file on storage medium that has been accidentally erased or destroyed.
32	MEDIUM FULL ► No available space on storage medium.
39	CONTROL LINES HELD LOW ► A device connected to the I/O bus is interfering with communication.
255	TIME-OUT ERROR ► Lost communication with the specified device. ► Specified device is not connected to the I/O bus.

Appendix E: Numeric Accuracy

The TI-74, like all 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.

Calculation Accuracy

The TI-74 uses a minimum of 13 digits to perform calculations. The results are rounded to 10 digits when displayed in the default display format. The computer's 5/4 rounding technique adds 1 to the least significant digit of the display if the next nondisplayed digit is five or more. If this digit is less than five, no rounding occurs. Without these extra digits, inaccurate results such as the following would frequently be displayed.

$1/3 \times 3 = .9999999999$

This result occurs because 1/3 is maintained as .3333333333 in the finite internal representation of a number. However, when $1/3 \times 3$ is rounded to 10 digits, the answer 1. is displayed.

The more complex mathematical functions are calculated using iterative and polynomial methods. 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. However, certain calculations may cause the unexpected appearance of these extra digits as shown below.

$2/3 = .66666666666667$ and $1/3 = .33333333333333$

$2/3 - 1/3 - 1/3 = .00000000000001$ (displayed 1.E-14)

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. In testing for equality, precautions should be taken to prevent improper evaluation.

Numeric Comparisons

A useful technique is to test whether two values are sufficiently close together rather than absolutely equal as shown below.

Instead of
```
IF X=Y THEN ...
```

use
```
IF ABS(X-Y) < 1E-11 THEN ...
```

Extremely Small Values

When working with values very close to zero, you may get unexpected results because any number whose absolute value is less than 1×10^{-128} becomes zero.

Instead of
```
SQR(3E-70^2+4E-70^2)
```

use
```
(1E-70*SQR(3^2+4^2))
```

Internal Numeric Representation

The TI-74 uses radix-100 format for internal calculations. Radix-100 makes possible the extensive numeric range of $-9.9999999999999E+127$ through $-1.E-128$; zero; and then $+1.E-128$ through $+9.9999999999999E+127$. Another benefit of this technique is 13, and sometimes 14, digits of internal precision.

Appendix F: Differences Between TI-74 BASIC and Others

This section is for programmers who have learned BASIC on another computer before using a TI-74. The lists below highlight features of TI-74 BASIC that may differ from the BASIC on an earlier machine.

Commands and Statements

You can use DISPLAY, as well as PRINT, to place information in the display. You usually need to include a PAUSE statement after a DISPLAY or PRINT statement so you can view the displayed information.

If you want a program to stop so you can check its progress and then resume execution, you should set a breakpoint. Refer to page 2-16.

Five instructions of TI-74 BASIC are available in abbreviated form.

CONTINUE	CON
DELETE	DEL
NUMBER	NUM
REM	!
RENUMBER	REN

The format string for a PRINT USING statement can be placed on a separate line. Refer to page 2-47.

The TI-74 has extra functions that enable you to calculate hyperbolic functions and their inverses.

You can include a quotation mark within a string by typing two consecutive quotation marks. Refer to page 1-9.

You can perform immediate calculations without using PRINT. Examples are shown in the *TI-74 User's Guide* on page 3-14.

Program Lines

The TI-74 protects program lines from accidental erasure by requiring the DELETE or NEW command. Entering only the line number and pressing **[ENTER]** does **not** delete the line.

Extra spaces are automatically deleted when you enter a program line. This conserves program memory and prevents the indentation of a program segment.

You can use the NUM command to automatically generate line numbers as you type a program.

A shortcut to listing a specific program line is the *line-number* **[↓]** key sequence.

Variables

All characters in a variable name are significant. Refer to page 1-4 for the rules of variable names.

A maximum of 95 variables can be assigned values concurrently.

The variables of a subprogram are separate from the variables of the main program even though they may have the same variable name. Refer to the page 2-121.

All numeric variables are full precision floating point. No variables are designated for integer or double precision.

Appendix F: Differences Between TI-74 BASIC and Others (Cont.)

Arrays

The TI-74 automatically defines all arrays as beginning with subscript 0.

If you enter a positive non-integer subscript, the subscript is interpreted as the integer nearest its value.

In one DIM statement, you can dimension several arrays. However, a line can contain only one DIM statement.

If you include a DIM statement in a multiple-statement line, it must be the last statement in the line.

The TI-74 automatically defines a default array if you use an array variable without first dimensioning the array. The default array has 11 elements (numbered 0 through 10) for each dimension that you specified in the variable.

Appendix G: Index

Use this list of items to find a topic of reference. Also see the Index of the TI-74 User's Guide.

Appendix G: Index

Appendix G: Index

Important Notice