ABS function--Absolute Value

Format

ABS(numeric-expression)

Type

Numeric (REAL or INTEGER)

Purpose

The ABS function returns the absolute value of the numeric-expression.

---

!o!     numeric-expression--If the value of the numeric-expression is positive or zero, ABS returns its value.  If the value of the numeric-expression is negative, ABS returns its negative (a positive number).

ABS always returns a non-negative number.

EXAMPLES
1100 PRINT ABS(42.3)

Prints 42.3.


|100 MAG=ABS(-6.124)

Sets MAG equal to 6.124.

## ACCEPT

### Format

ACCEPT [[AT(row,column)] [BEEP] [ERASE ALL] [SIZE(numeric-expression)]
    [VALIDATE(type[,...])]:]variable

### Purpose

The ACCEPT instruction suspends program execution to enable you to enter data
from the keyboard.

The options available with the ACCEPT instruction make it more versatile for
keyboard input than the INPUT statement.  You can accept up to one line of
input from any position within the screen window, sound a tone when the
computer is ready to accept input, clear the screen window before accepting
input, limit input to a specified number of characters, and define the types
of valid input.

You can use ACCEPT as either a program statement or a command.

### Cross Reference

GRAPHICS, INPUT, LINPUT, MARGINS, TERMCHAR

---

!o!    variable--The data value entered from the keyboard is assigned
to the variable you specify.  If you specify a numeric variable,
the data value entered from the keyboard must be a valid
representation of a number.  If you specify a string variable,
the data value entered from the keyboard can be either a string
or a number.  Trailing spaces are removed.

A string value entered from the keyboard can optionally be enclosed in
quotation marks.  However, a string containing a comma, a quotation mark, or
leading or trailing spaces must be enclosed in quotation marks.  A quotation
mark within a string is represented by two adjacent quotation marks.

You normally press ENTER to complete keyboard input; however, you can also use
AID, BACK, BEGIN, CLEAR, PROC'D, DOWN ARROW, or UP ARROW.  You can use the
TERMCHAR function to determine which of these keys was pressed to exit from
the previous ACCEPT, INPUT, or LINPUT instruction.

Note that pressing CLEAR during keyboard input normally causes a break in the
program.  However, if your program includes an ON BREAK NEXT statement, you
can use CLEAR to exit from an input field.

In High-Resolution and Multicolor Modes, data entered from the keyboard is not
visible on the screen.  ACCEPT cannot access the graphics portion of the
screen in the Split-Screen Modes.

Options

You can enter the following options, separated by a space, in any order.

!o!     AT--The AT option enables you to specify the location of the
        beginning of the input field. Row and column are relative to
        the upper-left corner of the screen window defined by the
        margins. The upper-left corner of the window defined by the
        margins is considered to be the intersection of row 1 and column
        1 by an ACCEPT instruction that uses the AT option. If you do
        not use the AT option, the input field begins in the far left
        column of the bottom row of the window.

!o!     BEEP--The BEEP option sounds a short tone to signal that the
        computer is ready to accept input.

!o!     ERASE ALL--The ERASE ALL option places a space character (ASCII
        code 32) in every character position in the screen window before
        accepting input. The graphics portion of the screen is not
        affected.

!o!     SIZE--The SIZE option enables you to specify a limit to the
        number of characters that can be entered as input. The limit is
        the absolute value of the numeric-expression. If the algebraic
        sign of the numeric-expression is positive, or if you do not use
        the SIZE option, the input field is cleared before input is
        accepted. If the numeric-expression is negative, the input
        field is not cleared, enabling you to place a value in the input
        field that may be accepted by pressing ENTER. If you do not use
        the SIZE option, or if the absolute value of the
        numeric-expression is greater than the number of characters
        remaining in the row (from the beginning of the input field to
        the right margin), the input field extends to the right margin.

!o!     VALIDATE--The VALIDATE option enables you to specify the
        characters or types of characters that are valid input. If you
        specify more than one type, a character from any of the
        specified types is valid. The types are as follows:

| TYPE | VALID INPUT |
| --- | --- |
| ALPHA | All alphabetic characters. |
| UALPHA | All upper-case alphabetic characters. |
| LALPHA | All lower-case alphabetic characters. |
| DIGIT | All digits (0-9). |
| NUMERIC | All digits (0-9), the decimal point (.), the plus sign (+), the minus sign (-), and the upper-case letter E. |

You can also use one or more string expressions as <u>types</u>. The characters contained in the strings specified by the string expressions are valid input.

The VALIDATE option only verifies data entered from the keyboard. If there is a default value in the input field (entered with DISPLAY, for example), the VALIDATE option has no effect on that value.

## EXAMPLES
|100 ACCEPT AT(5,7):Y

Accepts data at the fifth row, seventh column of the screen window into the variable Y.

|100 ACCEPT VALIDATE("YN"):R
$

Accepts data containing Y and/or N into the variable R$. (YYNN would be a valid entry.)

|100 ACCEPT ERASE ALL:B

Accepts data into the variable B after putting the blank character into all positions in the screen window.

|100 ACCEPT AT(R,C)SIZE(FIEL
DLEN)BEEP VALIDATE(DIGIT,"AY
N"):X$

Accepts a digit or the letters A, Y, or N into the variable X$. The length of the input may be up to FIELDLEN characters. A field the length of FIELDLEN is filled with blank characters, and then the data value is accepted at row R, column C. A beep is sounded before acceptance of data. (The beep option is especially useful in High-Resolution and Multicolor Modes, because the cursor is not visible.)

PROGRAM

The following program illustrates a typical use of ACCEPT. It allows entry of up to 20 names and addresses, and then displays them all.

This program will work in Pattern and Text Modes, but not in High-Resolution or Multicolor Modes.

```
|100 DIM NAME$(20),ADDR$(20)
|110 DISPLAY AT(5,1)ERASE AL
L:"NAME:"
|120 DISPLAY AT(7,1):"ADDRES
S:"
|130 DISPLAY AT(23,1):"TYPE
A ? TO END ENTRY."
|140 FOR S=1 TO 20
|150 ACCEPT AT(5,7)VALIDATE(
ALPHA,"?")BEEP SIZE(13):NAME
$(S)
|160 IF NAME$(S)="?" THEN 20
0
|170 ACCEPT AT(7,10)SIZE(12)
:ADDR$(S)
|180 DISPLAY AT(7,10):" "
|190 NEXT S
|200 CALL CLEAR
|210 DISPLAY AT(1,1):"NAME",
"ADDRESS"
|220 FOR T=1 TO S-1
|230 DISPLAY AT(T+2,1):NAME$
(T),ADDR$(T)
|240 NEXT T
|250 GOTO 250
(Press CLEAR to stop the program.)
```

To enter this example in the Split-Screen Modes, change line 130 to DISPLAY AT(8,1):"TYPE A ? TO END ENTRY". Remember that only 8 lines of output will be visible.

ASC Function—ASCII Value

**Format**

ASC(<u>string-expression</u>)

**Type**

INTEGER

**Purpose**

The ASC function returns the ASCII character code corresponding to the first character of the <u>string-expression</u>.

ASC is the inverse of the CHR$ function.

**Cross Reference**

CHR$

-----------------------------------------------------------------------

!o!    The <u>string-expression</u> cannot be a null string.

See Appendix A for a list of ASCII character codes.

<u>EXAMPLES</u>
|100 PRINT ASC("A")

Prints 65 (the ASCII character code for the letter A).

|100 B=ASC("1")

Sets B equal to 49 (the ASCII character code for the character 1).

|100 DISPLAY ASC("HELLO")

Displays 72 (the ASCII character code for the letter H).

|100 A$="DAVID"
|110 PRINT ASC(A$)

Prints 68 in line 110 .

ATN Function--Arctangent

Format

ATN(numeric-expression)

Type

REAL

Purpose

The ATN function returns the angle (in radians) whose tangent is the value of the numeric-expression.

Cross Reference

COS, SIN, TAN

---

The value returned by ATN is always greater than $-pi/2$ and less than $pi/2$.

To convert radians to degrees, multiply by 180/pi.

EXAMPLES
1100 PRINT 4*ATN(-1)

Prints -3.141592654.

|100 Q=PI/ATN(1.732)

Sets Q equal to 3.0000363894830.

BREAK

Format

BREAK [line-number-list]

Purpose

The BREAK instruction sets a breakpoint at each program statement you specify. When the computer encounters a line at which you have set a breakpoint, your program stops running before that statement is executed.

BREAK is a valuable debugging aid. You can use BREAK to stop your program at a specific program line, so that you can check the values of variables at that point.

You can use BREAK line-number-list as either a program statement or a command.

Cross Reference

CONTINUE, ON BREAK, UNBREAK

--------------------------------------------------------------------------------
!o!     line-number-list--The line-number-list consists of one or more
        line numbers, separated by commas. When a BREAK instruction is
        executed, breakpoints are set at the specified program lines.
        If you use BREAK as a program statement, the line-number-list is
        optional. When a BREAK statement with no line-number-list is
        encountered, the computer stops running the program at that
        point.

        If you use BREAK as a command, you must include a
        line-number-list.

Breakpoints

When your program stops at a breakpoint, the message Breakpoint in line-number is displayed. While your program is stopped at a breakpoint, you can enter any valid command.

To resume program execution starting with the line at which the break occurred, enter the CONTINUE command. However, if you edit your program (add, delete, or change a program statement) you cannot use CONTINUE. (This prevents errors that could result from resuming execution in the middle of a revised program.) You also cannot use CONTINUE if you enter a MERGE or SAVE command or a LIST command with the file-specification option.
Note that pressing CLEAR (FCTN 4) also causes a breakpoint to occur before the execution of the next program statement. When your program stops at a breakpoint, the computer performs the following operations:

!o!     It restores the default character definitions of all characters.

!o!     If the computer is in High-Resolution or Multicolor Mode, it
        restores the default graphics mode (Pattern) and margin settings
        (2, 2, 0, 0).

!o!     It restores the default foreground color (black) and background
        color (transparent) to all characters.

!o!     It restores the default screen color (cyan).

!o!     It deletes all sprites.

!o!     It resets the sprite magnification level to 1.

The graphics colors (see DCOLOR) and current position (see DRAWTO) are not
affected.  If the computer is in Pattern or Text Mode or one of the
Split-Screen Modes, the graphics mode and margin settings remain unchanged.


## Removing Breakpoints

You can remove a breakpoint by using the UNBREAK instruction or by editing or
deleting the line at which the breakpoint is set.  When your program stops at
a breakpoint, that breakpoint is automatically removed.

All breakpoints are removed when you use the NEW or SAVE command.

A BREAK statement with no line-number-list establishes a breakpoint that can
be removed only by deleting that program statement.

## BREAK Errors

If the line-number-list includes an invalid line number (0 or a value greater
than 32767), the message Bad line number is displayed.  If the
line-number-list includes a fractional or negative line number, the message
Syntax error is displayed.  In both cases, the BREAK instruction is ignored;
that is, breakpoints are not set even at valid line numbers in the
line-number-list.  If you were entering BREAK as a program statement it is not
entered into your program.

If the line-number-list includes a line number that is valid (1-32767) but is
not the number of a line in your program, or a fractional number greater than
1, the message

        * WARNING
          LINE NOT FOUND

is displayed.  (If you were entering BREAK as a program statement, the line
number is included in the warning message.)  A breakpoint is, however, set at
any valid line in the line-number-list preceding the line number which caused
the warning.

EXAMPLES
| 150 BREAK

BREAK as a statement causes a breakpoint before execution of the next line in the program.

|100 BREAK 120,130

Causes breakpoints before execution of lines 120 and 130.

|BREAK 10,400,130

As a command, causes breakpoints before execution of lines 10, 400, and 130.

BYE

Format

BYE

Purpose

The BYE command resets the computer.  Always use BYE to exit from TI Extended BASIC II.
_____
The BYE command causes the computer to do the following:

!o!    Close all open files.

!o!    Erase the program and all variable values in memory.

!o!    Exit from TI Extended BASIC II.

!o!    Display the master title screen.

Although you can exit from Extended BASIC II also by pressing QUIT (FCTN =), pressing QUIT does not close open files and may result in the loss of data in those files.

CALL

Format

CALL subprogram-name[(parameter-list)]

Purpose

The CALL instruction transfers program control to the specified subprogram.

You can use CALL as either a program statement or a command.

Cross Reference

SUB

---

!o!    subprogram-name--The CALL instruction transfers program control
       to the subprogram specified by the subprogram-name.

!o!    parameter-list--The optional parameter-list consists of one or
       more parameters separated by commas.  Use of a parameter-list is
       determined by the subprogram you are calling.  Some subprograms
       require a parameter-list, some do not use a parameter-list, and
       with some a parameter-list is optional.

You can use CALL as a program statement to call either a built-in TI Extended
BASIC II subprogram or to call a subprogram that you write.  After the
subprogram is executed, program control returns to the statement immediately
following the CALL statement.

You can use CALL as a command only to call a built-in Extended BASIC II
subprogram, not to call a subprogram that you write.

Each of the following built-in subprograms is discussed separately in this
manual.

        CHAR
        CHARPAT
        CHARSET
        CLEAR
        COINC
        COLOR
        DCOLOR
        DELSPRITE
        DISTANCE
        DRAW
        DRAWTO
        ERR
        FILL

GCHAR
GRAPHICS
HCHAR
INIT
JOYST
KEY
LINK
LOAD
HCHAR
LOCATE
MAGNIFY
MARGINS
.MOTION
PATTERN
PEEK
PEEKV
POKEV
POSITION
SAY
SCREEN
SOUND
SPGET
SPRITE
VCHAR
VERSION

## PROGRAM
The following program illustrates the use of CALL with a built-in subprogram (CLEAR) in line 100 and the use of a user-written subprogram (TIMES) in line 120.

```
|100 CALL CLEAR
|110 X=4
|120 CALL TIMES(X)
|130 PRINT X
|140 STOP
|200 SUB TIMES(Z)
|210 Z=Z*PI
|220 SUBEND
|RUN
(screen clears)
12.56637061
```

CHAR subprogram--Character Definition

## Format

CALL CHAR(<u>character-code</u>,<u>pattern-string</u>[,...])

## Purpose

The CHAR subprogram enables you to define your own characters so that you can create graphics on the screen.

CHAR is the inverse of the CHARPAT subprogram.

## Cross Reference

CHARPAT, CHARSET, COLOR, DCOLOR, GRAPHICS, HCHAR, SCREEN, SPRITE, VCHAR

------------------------------------------------------------------------

!o!    <u>character-code</u>--Character-code is a numeric expression with a value from 0 to 255, specifying the number of the character you want to define. You can define any of the 256 characters (codes 0-255), and display them as characters and/or sprites.

See Appendix A for a list of the available characters.

!o!    <u>pattern-string</u>--The <u>pattern-string</u> specifies the definition of the character. The <u>pattern-string</u>, which may be up to 64 digits long, is a coded representation of the pixels that define up to four characters on the screen, as explained below. Any letters entered as part of a <u>pattern-string</u> must be upper case.

You can use the CHARSET subprogram to restore default character definitions of characters 32-95 inclusive. Also, when your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode, all default character definitions (0-255) are restored.

The instructions that you can use to display characters on the screen vary according to the graphics mode. In all modes except Text Mode, you can use the SPRITE subprogram to display sprites on the screen.

If you use HCHAR or VCHAR to display a character on the screen and then later use CHAR to change the definition of that character, the result depends on the graphics mode.

!o!    In Pattern and Text Modes, the displayed character changes to the newly defined pattern.

!o!    In High-Resolution Mode and in the Split-Screen Modes, the displayed character remains unchanged.

Pattern, Split-Screen, and High-Resolution Modes

In Pattern and High-Resolution Modes and in the Split-Screen Modes, each character is composed of 64 pixels in a grid eight pixels high and eight pixels wide, as explained below.

In Pattern Mode, you can use the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions and the HCHAR and VCHAR subprograms to display characters on the screen.

In the Split-Screen Modes, you can use the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions to display characters in the text portion of the screen, and the HCHAR and VCHAR subprograms to display characters in the graphics portion of the screen.

In High-Resolution Mode, you can use the HCHAR and VCHAR subprograms to display characters on the screen.

Text Mode

In Text Mode, each character is composed of 48 pixels in a grid eight pixels high and six pixels wide. The eight by eight grid described below is used to define characters; however, the last two pixels in each pixel-row are ignored.

In Text Mode, you can use the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions and the HCHAR and VCHAR subprograms to display characters on the screen. You cannot display sprites in Text Mode.

Multicolor Mode

In Multicolor Mode, CHAR is useful only to define sprites. Each character used to define a sprite is composed of 64 pixels in a grid eight pixels high and eight pixels wide, as explained below.

Character Definition--The Pattern-String

Characters are defined by turning some pixels on and leaving others off. The space character (ASCII code 32) is a character with all the pixels turned off. Turning all the pixels on produces a solid block, eight pixels high and eight pixels wide.

The foreground color is the color of the pixels that are on. The background color is the color of the pixels that are off. (For more information see COLOR, DCOLOR, and SCREEN.)

When you enter TI Extended BASIC II, the characters are predefined with the appropriate pixels turned on. To redefine a character, you specify which pixels to turn on and which pixels to turn off.

For the purpose of defining characters, each pixel-row (eight pixels) is divided into two blocks (four pixels each). Each digit in the pattern-string is a code specifying the pattern of the four pixels in one block.

```
                              LEFT   £  RIGHT
                              BLOCKS £  BLOCKS
                                     £
                                     _____
            PIXEL-ROW 1       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 2       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 3       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 4       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 5       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 6       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 7       £_£_£_£_£_£_£_£_£
            PIXEL-ROW 8       £_£_£_£_£_£_£_£_£
```

You define a character by describing the blocks from left to right and from top to bottom. The first two digits in the pattern-string describe the pattern for the first two blocks (pixel-row 1) of the grid, the next two digits define the next two blocks (pixel-row 2), and so on.

The computer uses a binary (base 2) code to represent the status of each pixel; you use hexadecimal (base 16) notation of binary code to specify which pixels in a block are turned on and which pixels are turned off.

The following table shows all the possible on/off combinations of the four pixels in a block and the binary code and hexadecimal notation representing each combination.

| BLOCK | BINARY CODE (0=OFF; 1=ON) | HEXADECIMAL NOTATION |
|---|---|---|
| £_£_£_£_£ | 0000 | 0 |
| £_£_£_£*£ | 0001 | 1 |
| £_£_£*£_£ | 0010 | 2 |
| £_£_£*£*£ | 0011 | 3 |
| £_£*£_£_£ | 0100 | 4 |
| £_£*£_£*£ | 0101 | 5 |
| £_£*£*£_£ | 0110 | 6 |
| £_£*£*£*£ | 0111 | 7 |
| £*£_£_£_£ | 1000 | 8 |
| £*£_£_£*£ | 1001 | 9 |
| £*£_£*£_£ | 1010 | A |
| £*£_£*£*£ | 1011 | B |
| £*£*£_£_£ | 1100 | C |
| £*£*£_£*£ | 1101 | D |
| £*£*£*£_£ | 1110 | E |
| £*£*£*£*£ | 1111 | F |

A character definition consists of 16 hexadecimal digits; each digit represents one of the 16 blocks that comprise a character. As the pattern-string may be up to 64 digits long, you can define as many as four consecutive characters with one pattern-string.

If the length of the <u>pattern-string</u> is not a multiple of 16, the computer fills the <u>pattern-string</u> with zeros until its length is a multiple of 16.


## PROGRAMS

For the dot pattern pictured below, you use "1898FF3D3C3CE404" as the pattern string for CALL CHAR.

|  | LEFT BLOCKS | RIGHT BLOCKS | BLOCK CODES |
|---|---|---|---|
| ROW 1 | ! ! ! !*!*! ! ! ! | | 18 |
| ROW 2 | !*! ! !*!*! ! ! ! | | 98 |
| ROW 3 | !*!*!*!*!*!*!*!*! | | FF |
| ROW 4 | ! ! !*!*!*!*! !*! | | 3D |
| ROW 5 | ! ! !*!*!*!*! ! ! | | 3C |
| ROW 6 | ! ! !*!*!*!*! ! ! | | 3C |
| ROW 7 | !*!*!*! ! !*! ! ! | | E4 |
| ROW 8 | ! ! ! ! ! !*! ! ! | | 04 |

The following program uses this and one other string to make a figure "dance."

Note that this example will work only in Pattern Mode.

```
100 CALL CLEAR
110 A$="1898FF3D3C3CE404"
120 B$="1819FFBC3C3C2720"
130 CALL COLOR(27,7,12)
140 CALL VCHAR(12,16,244)
150 CALL CHAR(244,A$)
160 GOSUB 200
170 CALL CHAR(244,B$)
180 GOSUB 200
190 GOTO 150
200 FOR DELAY=1 TO 150
210 NEXT DELAY
220 RETURN
RUN
(screen clears)
(character moves)
```
        (Press <u>CLEAR</u> to stop the program.)

To make this example work in the Split-Screen and High-Resolution Modes, make
the following changes.

```
|105 CALL GRAPHICS(X)
  .
  .
|130 CALL DCOLOR(7,12)
|140 CALL CHAR(244,A$,245,B$
)
|150 CALL VCHAR(12,16,244)
  .
|170 CALL VCHAR(12,16,245)
  .
  .
  .
```

The X in line 105 must be replaced with the number of the graphics mode to be
designated (substitute a 3 or 4 for the Split-Screen Modes, and a 5 for
High-Resolution Mode).

If a program stops for a breakpoint, all characters are reset to their
standard patterns. When the program ends normally or because of an error, all
characters are reset.

Exiting High-Resolution or Multicolor Modes resets all characters.

The following example works in all graphics modes except Multicolor Mode.

```
|100 CALL CLEAR
|110 CALL GRAPHICS(X)
|120 CALL CHAR(244,"FFFFFFFF
FFFFFFFF")
|130 CALL CHAR(42,"0F0F0F0F0
F0F0F0F")
|140 CALL HCHAR(12,17,42)
|150 CALL VCHAR(14,17,244)
|160 FOR DELAY=1 TO 500
|170 NEXT DELAY
|RUN
```

The X in line 110 must be replaced with the number of the graphics mode to be
designated.

CHARPAT subprogram--Character Pattern

Format

CALL CHARPAT(character-code,string-variable[,...])

Purpose

The CHARPAT subprogram enables you to ascertain the current character definitions of specified characters.

Cross Reference

CHAR

---

!o!    character-code--Character-code is a numeric expression with a value from 0 to 255, specifying the number of the character of which you want the current definition.

!o!    string-variable--The pattern describing the character definition is returned in the specified string-variable. The pattern is in the form of a 16-digit hexadecimal code. See CHAR for an explanation of the patterns used for character definition.

See Appendix A for a list of the available characters.

EXAMPLE
T100 CALL CHARPAT(33,C$)

Sets C$ equal to "0010101010001000", the pattern identifier for character 33, the exclamation point.

CHARSET subprogram--Set Characters

**Format**

CALL CHARSET

**Purpose**

The CHARSET subprogram restores the default character definitions and colors.

**Cross Reference**

CHAR, COLOR

--------------------------------------------------------------------------

CHARSET restores the default character definitions to characters 32-95, inclusive.  CHARSET restores the default colors to all 256 characters.

See Appendix A for a list of the available characters.

CHR$ function--Character

Format

CHR$(character-code)

Type

String

Purpose

The CHR$ function returns the character corresponding to the ASCII character code specified by the value of the character-code.

CHR$ is the inverse of the ASC function.

Cross Reference

ASC

------------------------------------------------------------------------

!o!     character-code--Character-code is a numeric expression with a value from 0 to 32767 inclusive, specifying the number of the character you wish to use. If the value of character-code is greater than 255, it is repeatedly reduced by 256 until it is less than 256. If the value of the character-code is not an integer, it is rounded to the nearest integer.

See Appendix A for a list of ASCII character codes.

EXAMPLES
100 PRINT CHR$(72)

Prints H.

100 X$=CHR$(33)

Sets X$ equal to !.

## PROGRAM
For a complete list of all ASCII characters and their corresponding ASCII values, run the following program.

```
|100 CALL CLEAR
|110 IMAGE ### ## ### ##
|120 FOR A=32 TO 127
|130 PRINT USING 110:A,CHR$(
A);
|140 NEXT A
|150 GOTO 150
        (Press CLEAR to stop the program.)
```

CLEAR subprogram

**Format**

CALL CLEAR

**Purpose**

The CLEAR subprogram erases the screen.

**Cross Reference**

DCOLOR, DELSPRITE

---

In Pattern and Text Modes and in the text portion of the screen in the Split-Screen Modes, CLEAR places a space character (ASCII code 32) in every screen position.

In High-Resolution Mode and in the graphics portion of the screen in the Split-Screen Modes, CLEAR erases the screen by turning off all pixels and restoring the default graphics colors (black on transparent).

In Multicolor Mode, CLEAR sets the color of each block to transparent.

The CLEAR subprogram has no effect on sprites. Use the DELSPRITE subprogram to remove sprites.

PROGRAMS
When the following program is run, the screen is cleared before the PRINT statements are performed.

```
|100 CALL CLEAR
|110 PRINT "HELLO THERE!"
|120 PRINT "HOW ARE YOU?"
|RUN
--screen clears
HELLO THERE!
HOW ARE YOU?
```

If the space character (ASCII code 32) has been redefined by the CALL CHAR subprogram, the screen is filled with the new character when CALL CLEAR is performed.

```
|100 CALL CHAR(32,"0103070F1
F3F7FFF")
|110 CALL CLEAR
|120 GOTO 120
|RUN
-- screen is filled with *
        (Press CLEAR to stop the program.)
```

The following program clears even the graphics portion of the screen.

```
|100 CALL GRAPHICS(5)
|110 CALL HCHAR(1,1,72,768)
|120 FOR DELAY=1 TO 500::NEX
T DELAY
|130 CALL CLEAR
|140 GOTO 140
|RUN
        (Press CLEAR to stop the program.)
```

CLOSE

## Format

CLOSE #file-number[:DELETE]

## Purpose

The CLOSE instruction closes the specified file.  When you close a file, you discontinue the association (between your program and the file) that you established in an OPEN instruction.

You can use CLOSE as either a program statement or a command.

## Cross Reference

DELETE, OPEN

---

!o!　　file-number--The file-number is a numeric expression whose value specifies the number of the file as assigned in its OPEN instruction.

!o!　　DELETE--The DELETE option, which can be used only with certain devices, deletes the file after closing it.  DELETE has no effect on a file stored on an audio cassette.  For more information about using the DELETE option with a particular device, refer to the owner's manual that comes with that device.

After the CLOSE instruction is performed, the closed file cannot be accessed by an instruction, because the computer no longer associates that file with a file-number.  You can then reassign the file-number to another file.

When you close a file on a cassette, the computer displays instructions for you to follow.

Closing Files without the CLOSE Instruction

To protect the data in your files, the computer closes all open files when it reaches the end of your program or when it encounters an error (either in Command or Run mode).

Open files are also closed when you do one of the following:

!o!　　Edit your program (add, delete, or change a program statement).

!o!　　Enter the LIST command with the file-specification option.

!o!　　Enter the BYE, MERGE, NEW, OLD, RUN, or SAVE command.

Always use BYE to exit from Extended BASIC II.  Although you can also exit by pressing QUIT (FCTN =), pressing QUIT does not close open files, and may result in the loss of data in those files.

Open files are not closed when you stop program execution by pressing CLEAR (FCTN 4) or when your program stops at a breakpoint set by a BREAK instruction.


EXAMPLES

When the computer performs the CLOSE statement for a cassette tape recorder, you receive instructions for operating the recorder.  The following two examples show the difference between closing a cassette file and closing a diskette file.

Cassette File

```
|100 OPEN #24:"CS1",INTERNAL
,OUTPUT,FIXED
 •
 •       (program lines)
 •
|200 CLOSE #24
|RUN
REWIND CASSETTE TAPE
THEN PRESS ENTER
 •
PRESS CASSETTE RECORD
THEN PRESS ENTER
 •
 •       (program runs)
 •
PRESS CASSETTE STOP
THEN PRESS ENTER
```

Diskette File

```
|100 OPEN #24:"DSK1.MYDATA",
INTERNAL,UPDATE,FIXED
 •
 •       (program lines)
 •
|200 CLOSE #24
|RUN
 •
 •       (program runs)
 •
```

The CLOSE statement for a diskette requires no further action on your part.

COINC subprogram--Coincidence

Format

Two Sprites
        CALL COINC(#sprite-number1,#sprite-number2,tolerance,numeric-variable)
A Sprite and a Screen Pixel
        CALL COINC(#sprite-number,pixel-row,pixel-column,tolerance,
            numeric-variable)

All Sprites
        CALL COINC(ALL,numeric-variable)

Purpose

The COINC subprogram enables you to ascertain if sprites are coincident (in conjunction) with each other or with a specified screen pixel.

Cross Reference

SPRITE

---

The exact conditions that constitute a coincidence vary depending on  whether you are testing for the coincidence of two sprites, a sprite and a screen pixel, or all sprites.

If the sprites are moving very quickly, COINC may occasionally fail to detect a coincidence.

Two Sprites

Two sprites are considered to be coincident if the upper-left corners of the sprites are within a specified number of pixels (tolerance) of each other.

> !o!    sprite-numbers--The values of the numeric expressions
>        sprite-number1 and sprite-number2 specify the numbers of the two
>        sprites as assigned in the SPRITE subprogram.

> !o!    tolerance--A coincidence exists if the distance between the
>        pixels in the upper-left corners of the two sprites is less than
>        or equal to the value of the numeric expression tolerance.

>        The distance between two pixels is said to be within tolerance
>        if the difference between pixel-rows and the difference between
>        pixel-columns are both less than or equal to the specified
>        tolerance.  Note that this is not the same as the distance
>        indicated by the DISTANCE subprogram.

!o!  numeric-variable--COINC returns a value in the numeric-variable
     indicating whether or not the specified coincidence exists.  The
     value is -1 if there is a coincidence or 0 if there is no
     coincidence.

## A Sprite and a Screen Pixel

A sprite is considered to be coincident with a screen pixel if the upper-left
corner of the sprite is within a specified number of pixels (tolerance) of the
screen pixel or if any pixel in the sprite occupies the screen pixel location.

!o!  sprite-number--The sprite-number is a numeric expression whose
     value specifies the number of the sprite as assigned in the
     SPRITE subprogram.

!o!  pixel-row and pixel-column--The pixel-row and pixel-column are
     numeric expressions whose values specify the position of the
     screen pixel.

!o!  tolerance--A coincidence exists if the distance between the
     pixel in the upper-left corner of the sprite and the screen
     pixel is less than or equal to the value of the numeric
     expression tolerance.  (Note that a coincidence also exists if
     any pixel in the sprite occupies the screen pixel location.)

     The distance between two pixels is said to be within tolerance
     if the difference between pixel-rows and the difference between
     pixel-columns are both less than or equal to the specified
     tolerance.  Note that this is not the same as the distance
     indicated by the DISTANCE subprogram.

!o!  numeric-variable--COINC returns a value in the numeric-variable
     indicating whether or not the specified coincidence exists.  The
     value is -1 if there is a coincidence or 0 if there is no
     coincidence.

## All Sprites

The ALL option tests for the coincidence of any of the sprites.

!o!  ALL--For the ALL option, sprites are considered to be coincident
     if any pixel of any sprite occupies the same screen pixel
     location as any pixel of any other sprite.

!o!  numeric-variable--COINC returns a value in the numeric-variable
     indicating whether or not a coincidence exists.  The value is -1
     if there is a coincidence or 0 if there is no coincidence.

## PROGRAM
The following program defines two triangular sprites.

```
|100 CALL CLEAR
|110 S$="0103070F1F3F7FFF"
|120 CALL CHAR(244,S$)
|130 CALL CHAR(250,S$)
|140 CALL SPRITE(#1,244,7,50
,50)
|150 CALL SPRITE(#2,250,5,44
,42)
|160 CALL COINC(#1,#2,10,C)
|170 PRINT C
|180 CALL COINC(ALL,C)
|190 PRINT C
|200 GOTO 200
|RUN
-1
 0
```

        (Press CLEAR to stop the program.)

Line 160 shows a coincidence because the upper-left corners of the sprites are within 10 pixels of each other.

Line 180 shows no coincidence because the shaded areas of the sprites do not occupy the same screen pixel location. (Shaded areas are compared only if you specify the ALL option.)

COLOR subprogram


Format

Pattern Mode
        CALL COLOR(character-set,foreground-color,background-color[,...])
Split-Screen Modes
        CALL COLOR(character-code,foreground-color,background-color[,...])
Multicolor Mode
        CALL COLOR(row,column,color[,...])
Sprites
        ·CALL COLOR(#sprite-number,foreground-color[,...])


Purpose

The COLOR subprogram enables you to specify the colors of characters, blocks, or sprites.

Cross Reference

CHAR, DCOLOR, GRAPHICS, SCREEN, SPRITE

------------------------------------------------------------------------

The types of parameters you specify in a call to the COLOR subprogram depend on the graphics mode.  There are also different parameters for assigning colors to sprites.

In Pattern Mode and in the text portion of the screen in the Split-Screen Modes, each character has two colors.  The color of the pixels that make up the character itself is the foreground-color; the color of the pixels that occupy the rest of the character position on the screen is the background-color.

When you enter TI Extended BASIC II, the foreground-color of all characters is black; the background-color of all characters is transparent.  These default colors are restored when your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode.

If a color is transparent, the color actually displayed is the color specified by the SCREEN subprogram.

The codes for the available colors are listed below and in Appendix J.

| CODE | COLOR |
|------|-------|
| 1 | Transparent |
| 2 | Black |
| 3 | Medium Green |
| 4 | Light Green |
| 5 | Dark Blue |
| 6 | Light Blue |
| 7 | Dark Red |
| 8 | Cyan |
| 9 | Medium Red |
| 10 | Light Red |
| 11 | Dark Yellow |
| 12 | Light Yellow |
| 13 | Dark Green |
| 14 | Magenta |
| 15 | Gray |
| 16 | White |

See Appendix K for a list of recommended color combinations.

Pattern Mode

In Pattern Mode, the 256 available characters are divided into 32 sets of 8 characters each. When you assign a color combination to a particular set, you specify the colors of all 8 characters in that set.

!o!     character-set--The character-set is a numeric expression whose value specifies the number (0-31) of the 8-character set.

!o!     foreground-color, background-color--Foreground-color and background-color are numeric expressions whose values specify colors that can be assigned from among the 16 available colors.

The available characters and character sets in Pattern Mode are listed below and in Appendix E.

| SET | CHARACTER CODES |
|---|---|
| 0 | 24-31 |
| 1 | 32-39 |
| 2 | 40-47 |
| 3 | 48-55 |
| 4 | 56-63 |
| 5 | 64-71 |
| 6 | 72-79 |
| 7 | 80-87 |
| 8 | 88-95 |
| 9 | 96-103 |
| 10 | 104-111 |
| 11 | 112-119 |
| 12 | 120-127 |
| 13 | 128-135 |
| 14 | 136-143 |
| 15 | 144-151 |
| 16 | 152-159 |
| 17 | 160-167 |
| 18 | 168-175 |
| 19 | 176-183 |
| 20 | 184-191 |
| 21 | 192-199 |
| 22 | 200-207 |
| 23 | 208-215 |
| 24 | 216-223 |
| 25 | 224-231 |
| 26 | 232-239 |
| 27 | 240-247 |
| 28 | 248-255 |
| 29 | 0-7 |
| 30 | 8-15 |
| 31 | 16-23 |

Text Mode

An error occurs if you use the COLOR subprogram to assign character colors in Text Mode. Use the SCREEN subprogram to assign character colors in Text Mode.

In Text Mode, using the COLOR program to assign colors to sprites has no effect (Text Mode does not display sprites).

Split-Screen Modes

In the Split-Screen Modes, you can use the COLOR subprogram to assign colors to characters in the text portion of the screen. Use the DCOLOR subprogram to specify character and graphics colors in the graphics portion of the screen.

!o!     character-code--Character-code is a numeric expression with a
        value from 0 to 255, specifying the number of the character .
        See Appendix A for a list of ASCII character codes.

!o!     foreground-color, background-color--Foreground-color and
        background-color are numeric expressions whose values specify
        colors that can be assigned from among the 16 available colors.

## High-Resolution Mode

In High-Resolution Mode, you can use COLOR only to assign colors to sprites;
any other use of the COLOR subprogram causes an error.  Use the DCOLOR
subprogram to specify character and graphics colors in the graphics portion of
the screen.

## Multicolor Mode

!o!     row, column--Row and column are numeric expressions whose values
        specify the screen location of the block.  The value of the
        numeric expression row must be greater than or equal to 1 and
        less than or equal to 48.  The value of the numeric expression
        column must be greater than or equal to 1 and less than or equal
        to 64.

!o!     color--Color is a numeric expression whose value specifies a
        color that can be assigned from among the 16 available colors.
        Each block can be assigned one color.

## Sprites

A sprite is assigned a foreground-color when it is created with the SPRITE
subprogram.  The background-color of a sprite is always transparent.

To re-assign colors to sprites you must use the sprite parameters, no matter
what graphics mode the computer is in.

!o!     sprite-number--The sprite-number is a numeric expression whose
        value specifies the number of a sprite as assigned by the SPRITE
        subprogram.

!o!     foreground-color--Foreground-color is a numeric expression whose
        value specifies a color that can be assigned from among the 16
        available colors.

EXAMPLES
|100 CALL COLOR(#5,16)

Sets sprite number 5 to have a <u>foreground-color</u> of 16 (white).  The <u>background-color</u> is always 1 (transparent).

This example is valid in all graphics modes.  (Remember that sprites have no effect in Text Mode.)


|100 CALL COLOR(#7,INT(RND*1
6+1))

Sets sprite number 7 to have a <u>foreground-color</u> chosen randomly from the 16 colors available.  The <u>background-color</u> is 1 (transparent).

This example is valid in all graphics modes.


PROGRAMS
In Pattern Mode, this program sets the <u>foreground-color</u> of characters 48-55 to 5 (dark blue) and the <u>background-color</u> to 12 (light yellow).

|100 CALL CLEAR
|110 CALL GRAPHICS(X)
|120 CALL COLOR(3,5,12)
|130 DISPLAY AT(12,16):CHR$
(48)
|140 GOTO 140
(Press <u>CLEAR</u> to stop the program.)

In the Split-Screen Modes, this program sets <u>character-code</u> 3 to dark blue on light yellow.  To show this, line 130 should be changed to read DISPLAY AT(12,16):CHR$(3).

The X in line 110 must be replaced with the number of the graphics mode to be designated.


In Multicolor Mode, the next program turns the block located at row 20, column 10 to yellow.

|100 CALL GRAPHICS(6)
|110 CALL COLOR(20,10,12)
|120 GOTO 120
        (Press <u>CLEAR</u> to stop the program.)

The next program is valid in both Split-Screen Modes.  CALL COLOR(72,7,12) sets character code 72 (H) to have a _foreground-color_ of red and a _background-color_ of yellow.

```
|100 CALL GRAPHICS(X)
|110 CALL COLOR(72,7,12)
|120 PRINT CHR$(72)
|130 GOTO 130
        (Press CLEAR to stop the program.)
```

The X in line 100 must be replaced with the number of the graphics mode to be designated (3 or 4).

CONTINUE

Format

;§CONTINUE§;
;§CON§;

Purpose

The CONTINUE command resumes the execution of a program that stopped because either a breakpoint was encountered or <u>CLEAR</u> (FCTN 4) was pressed. Program execution continues from the point where the break occurred.

Cross Reference

BREAK

---

The CONTINUE command causes your program to proceed as if no break had occurred. However, CONTINUE does not reverse the following effects of a breakpoint:

!o! Characters and colors are restored to their default definitions.

!o! All sprites are deleted and sprite magnification is reset.

!o! Graphics mode and margin settings may be reset (see BREAK).

If you edit your program (add, delete, or change a program statement) while it is stopped at a breakpoint (or after pressing <u>CLEAR</u>), you cannot use the CONTINUE command. This prevents errors that could result from resuming execution in the middle of a revised program.

You also cannot use CONTINUE if you enter a MERGE or SAVE command or a LIST command with the file-specification option.

COS function—Cosine

Format

COS(numeric-expression)

Type

REAL

Purpose

The COS function returns the cosine of the angle whose measurement in radians is the value of the numeric-expression.

Cross Reference

ATN, SIN, TAN

---

!o!   numeric-expression—The value of the numeric-expression cannot be less than -1.5707963269514E10 or greater than 1.5707963266374E10.

To convert the measure of an angle from degrees to radians, multiply by pi/180.

PROGRAM
The following program gives the cosine for each of several angles.

```
|100 A=1.047197551196
|110 B=60
|120 C=45*PI/180
|130 PRINT COS(A);COS(B)
|140 PRINT COS(B*PI/180)
|150 PRINT COS(C)
|RUN
  .5 -.9524129804
  .5
  .7071067812
```

DATA

Format

DATA data-list

Purpose

The DATA statement enables you to store constants within your program. You can assign the constants to variables by using a READ statement.

Cross Reference

READ, RESTORE

------------------------------------------------------------------------

!o!    data-list--The data-list consists of one or more constants
       separated by commas. The constants can be assigned to the
       variables specified in the variable-list of a READ statement.
       The assignment is made when the READ statement is executed.

If a numeric variable is specified in the variable-list of a 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 a READ statement, either a string or a numeric constant may be in the corresponding position in the DATA statement. A string constant in a data-list may optionally be enclosed in quotation marks. However, if the string constant contains a comma, a quotation mark, or leading or trailing spaces, it must be enclosed in quotation marks.

A quotation mark within a string constant is represented by two adjacent quotation marks. A null string is represented in a data-list by two adjacent commas, or two commas separated by two adjacent quotation marks.

The order in which the data values appear within the data-list and the order of the DATA statements within a program normally determine the order in which the values are read. Values from each data-list are read sequentially, beginning with the first item in the first DATA statement. If your program includes more than one DATA statement, the DATA statements are read in ascending line-number order (unless you use a RESTORE statement to specify otherwise).

A DATA statement encountered during program execution is ignored.

A DATA statement cannot be part of a multiple-statement line, nor can it include a trailing remark.

## PROGRAM
The following program reads and prints several numeric and string constants.

```
|100 FOR A=1 TO 5
|110 READ B,C
|120 PRINT B;C
|130 NEXT A
|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"

|190 FOR A=1 TO 6
|200 READ B$
|210 PRINT B$
|220 NEXT A
|230 DATA 1,NUMBER,TI
|RUN
  2   4
  6   7
  8   1
  2   3
  4   5
"THIS HAS QUOTES"
NO QUOTES HERE
   NO QUOTES HERE, EITHER
 1
NUMBER
TI
```

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

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

DCOLOR subprogram--Draw Color

**Format**

CALL DCOLOR(foreground-color,background-color)

**Purpose**

The DCOLOR subprogram enables you to set the graphics colors.

The graphics colors are used by the DRAW, DRAWTO, FILL, HCHAR, and VCHAR subprograms in High-Resolution Mode and in the graphics portion of the screen in the Split-Screen Modes.

**Cross Reference**

COLOR, DRAW, DRAWTO, FILL, GRAPHICS, HCHAR, VCHAR

---

!o!    foreground-color, background-color--Foreground-color and background-color are numeric expressions whose values specify colors that can be assigned from among the 16 available colors. See Appendix J for a list of the available colors.

When you enter TI Extended BASIC II, the foreground-color is set to black and the background-color is set to transparent. These default graphics colors are restored only when you change graphics mode. They are not restored when you enter RUN.

DCOLOR is effective only in High-Resolution Mode and in the graphics portion of the screen in the Split-Screen Modes. DCOLOR has no effect in Pattern, Text, or Multicolor Mode or in the text portion of the screen in the Split-Screen Modes.

**PROGRAMS**

The following program sets the foreground-color of graphics to 5 (dark blue) and the background-color to 8 (cyan).

```
|100 CALL CLEAR
|110 CALL GRAPHICS(X)
|120 CALL DCOLOR(5,8)
|130 CALL HCHAR(8,20,72,3)
|140 GOTO 140
        (Press CLEAR to stop the program.)
```

The X in line 100 must be replaced with the number of the graphics mode to be designated (3, 4, or 5).

In the following program, the letters "HHH" are displayed on the screen.

```
|100 CALL CLEAR
|110 CALL GRAPHICS(X)
|120 RANDOMIZE
|130 CALL DCOLOR(INT(RND*8+1
)*2,INT(RND*8+1)*2-1)
|140 CALL HCHAR(8,20,72,3)
|150 FOR X=1 TO 400
|160 NEXT X
|170 GOTO 120
        (Press CLEAR to stop the program.)
```

Line 130 changes the foreground-color (chosen randomly from the even-numbered colors available) and the background-color (chosen randomly from the odd-numbered colors). The X in line 100 must be replaced with the number of the graphics mode to be designated (3, 4, or 5).

DEF--Define Function

**Format**

DEF [data-type ]function-name[([data-type1 ]parameter1
    [,. . .[data-type7 ]parameter7])]=expression

**Purpose**

The DEF statement enables you to define your own functions. These
user-defined functions can then be used in the same way as built-in functions.

------------------------------------------------------------------------

!o! function-name--The function-name can be any valid variable name
that does not appear as a variable name elsewhere in your
program.

!o! expression--If the function-name is a numeric variable, the
value of the expression must be a number. If the function-name
is a string variable, the value of the expression must be a
string.

!o! data-type--If the function-name is a numeric variable, you can
optionally specify its data-type (INTEGER or REAL).

!o! parameters--You can use up to seven parameters to pass values to
a function. Parameters must be valid variable names. A
variable name used as a parameter cannot be the name of an
array. You can use an array element in the expression if the
array does not have the same name as a parameter in that
statement. The variable names used as parameters in a DEF
statement are local to that statement; that is, even if a
parameter has the same name as a variable in your program, the
value of that variable is not affected.

!o! parameter data-types--If a parameter is a numeric variable, you
can optionally specify its data-type (INTEGER or REAL).

A DEF statement must have a lower line number than that of any use of the
function-name it defines. A DEF statement is not executed.

A DEF statement can appear anywhere in your program, except that it cannot be
part of an IF THEN statement.

## DEF without <u>Parameters</u>

When your program encounters a statement containing a previously defined <u>function-name</u> with no <u>parameters</u>, the <u>expression</u> is evaluated, and the <u>function</u> is assigned the value of the <u>expression</u> at that time.

If you define a <u>function-name</u> without <u>parameters</u>, it must appear without <u>parameters</u> when you use it in your program.

## DEF with <u>Parameters</u>

When your program encounters a statement containing a previously defined <u>function-name</u> with <u>parameters</u>, the <u>parameter</u> values are passed to the function in the same order in which they are listed. The <u>expression</u> is evaluated using those values, and the function is assigned the value of the <u>expression</u> at that time. String values can be passed only to string <u>parameters</u>. Numeric values can be passed only to numeric <u>parameters</u>.

If you define a function with <u>parameters</u>, it must appear with the same number of <u>parameters</u> when you use it in your program.

## Recursive Definitions

A DEF statement may reference other defined functions (the <u>expression</u> may include previously defined <u>function-names</u>). However, a DEF statement may not be either directly or indirectly recursive (self-referencing).

Direct recursion occurs when you use the <u>function-name</u> in the <u>expression</u> of the same DEF statement. (This would be similar to writing a dictionary definition that included the word you were trying to define.)

Indirect recursion occurs when the <u>expression</u> contains a <u>function-name</u>, and in turn the <u>expression</u> in the DEF statement of that function (or other function subsequently referenced) includes the original <u>function-name</u>. (This would be similar to looking up the dictionary definition of a word, finding that the definition included other words that you needed to look up, and then discovering that the definitions led you directly back to your original word.)

## EXAMPLES

```
100 DEF PAY(OT)=40*RATE+1.5
*RATE*OT
110 RATE=4.00
120 PRINT PAY(3)
RUN
 178
```

Defines PAY so that each time it is encountered in a program the pay is figured using the RATE of pay times 40 plus 1.5 times the rate of pay times the overtime hours.

```
|100 DEF RND20=INT(RND*20+1)
```

Defines RND20 so that each time it is encountered in a program an integer from 1 through 20 is given.

```
|100 DEF FIRSTWORD$(NAME$)=S
EG$(NAME$,1,POS(NAME$," ",1)
-1)
```

Defines FIRSTWORD$ to be the part of NAME$ that precedes a space.

## PROGRAMS
The following program illustrates. a use of DEF.

```
|100 DEF A(INTEGER B)=SQR(B
)*5
|110 INPUT C
|120 PRINT A(C)
```

In line 100, the parameter B is assigned the INTEGER data-type.

In line 110, the value assigned to C is passed to the parameter B.

The following program does modulo arithmetic by using the user-defined function MOD. MOD accepts two parameters that are whole numbers.

```
|100 DEF MOD(X,Y)=X-(Y*INT(A
BS(X)/ABS(Y))*SGN(X*Y))
|110 PRINT MOD(3,2)
|120 PRINT MOD(500,3)
|130 PRINT MOD(25,5)
|140 PRINT MOD(25,3)
|RUN
  1
  2
  0
  1
```

DELETE

Format

DELETE file-specification

Purpose

The DELETE instruction removes a file from an external storage device.
Although the file is not physically erased, the space it occupies becomes
available for you to store another file in the future.

You can use DELETE as either a program statement or a command.

Cross Reference

CLOSE

---

!o!   file-specification--The file-specification indicates the name of
      the file to be deleted (see "File Specifications," beginning on
      page XX). The file-specification is a string expression; if you
      use a string constant, you must enclose it in quotation marks.

DELETE has no effect on a file stored on an audio cassette.

You can also remove files stored on some external devices by using the DELETE
option in the CLOSE instruction.

For more information about the options available with a particular device,
refer to the owner's manual that comes with that device.

EXAMPLE
|DELETE "DSK1.MYFILE"

Deletes the file named MYFILE from the diskette in disk drive 1.

PROGRAM
The following program illustrates a use of DELETE.

```
|100 INPUT "NAME OF FILE TO
BE DELETED: ":X$
|110 DELETE X$
```

DELSPRITE subprogram--Delete Sprite

**Format**

Delete Specified Sprites
        CALL DELSPRITE(#sprite-number[,...])
Delete All Sprites
        CALL DELSPRITE(ALL)

**Purpose**

The DELSPRITE subprogram enables you to delete one or more sprites.

**Cross Reference**

CLEAR, SPRITE

-------------------------------------------------------------------------
All sprites are deleted when your program ends (either normally or because of
an error), stops at a breakpoint, or changes graphics mode.

Delete Specified Sprites

!o!    sprite-number--Sprite-number is a numeric expression whose valu
       specifies the number of the sprite as assigned in the SPRITE
       subprogram.  The sprite can reappear if it is redefined by the
       SPRITE subprogram, or if the LOCATE subprogram is called.

Delete All Sprites

!o!    ALL--If you enter the ALL option, all sprites are deleted, and
       can reappear only if redefined by the SPRITE subprogram.

EXAMPLES
†100 CALL DELSPRITE(#3)

Deletes sprite number 3.

†100 CALL DELSPRITE(#4,#3*C)

Deletes sprite number 4 and the sprite whose number is found by multiplying :
by C.

|100 CALL DELSPRITE(ALL)

Deletes all sprites.

## DIM--Dimension

### Format

;§DIM        §;array-name(integer1[,... integer7])[,array-name... ]
;§data-type§;

### Purpose

The DIM instruction enables you to dimension (reserve space for) arrays with one to seven dimensions.

You can use DIM as either a program statement or a command.

### Cross Reference

INTEGER, OPTION BASE, REAL

------------------------------------------------------------------------

!o!   array-name--The array-name must be a valid variable name.  It cannot be used as the name of a variable or as the name of another array.  An array is either numeric or string, depending on the array-name.

!o!   The integer is the upper limit of element numbers in a dimension.

If a program includes an OPTION BASE 1 statement, the first element is element 1, so the number of elements is equal to the integer.

If a program does not include an OPTION BASE 1 statement, the first element is element 0, so the number of elements is equal to the integer plus 1.

A string array cannot have more than 16383 elements.  For numeric arrays, an INTEGER array cannot have more than 32767 elements, and a REAL array cannot have more than 8191 elements. The number of integers in parentheses following the array-name determines the number of dimensions (1-7) in the array.

!o!   data-type--You can optionally specify the data-type (INTEGER or REAL) of a numeric array by replacing DIM with the data-type.

An error occurs if you try to dimension a particular array more than once.

Note that you cannot use both instruction formats (DIM and data-type) to dimension the same array.

You cannot use OPTION BASE as a command.

You can dimension as many arrays with one DIM instruction as you can fit in one input line.

If you reference an array without first using a DIM instruction to dimension it, each dimension is assumed to have 11 elements (elements 0-10), or 10 elements (elements 1-10) if your program includes an OPTION BASE 1 statement.

If you use a DIM statement to dimension an array, the DIM statement must have a line number lower than that of any reference to that array. DIM statements are interpreted during pre-scan and are not executed.

A DIM statement can appear anywhere in your program, except as part of an IF THEN statement.

Referencing an Array

To reference a specific element of an array, you must use subscripts. Subscripts are numeric expressions enclosed in parentheses immediately following the reference to the array-name. An array reference must include one subscript for each dimension in the array.

If necessary, the value of a subscript is rounded to the nearest integer.

Reserving Space for Arrays

When you use DIM as a program statement, the computer reserves space for arrays when you enter the RUN instruction, before your program is actually run. If the computer cannot reserve space for an array with the dimensions you specify, the message Memory full in line-number is displayed, and your program does not run.

When you use DIM as a command, if the computer cannot reserve space for an array with the dimensions you specify, the message Memory full is displayed, and the command does not execute.

Until you place values in an array, each element in a string array is a null string and each element in a numeric array has a value of zero.

EXAMPLES

|100 DIM X$(30)

Reserves space in the computer's memory for 31 members of the array called X$.

|100 DIM D(100),B(10,9)

Reserves space in the computer's memory for 101 members of the array called D and 110 (11 times 10) members of the array called B.

|100 INTEGER B(10)

Reserves space in the computer's memory for 11 members of the array called B. However, INTEGER specifies that the members can only be integers.

# DISPLAY

## Format

DISPLAY [print-list]
DISPLAY [AT(row,column)] [BEEP] [ERASE ALL] [SIZE(numeric-expression)]
        [:print-list]

## Purpose

The DISPLAY instruction enables you to display numbers and strings on the screen. The numeric and/or string expressions in the print-list can be constants and/or variables.

The options available with the DISPLAY instruction make it more versatile for screen output than is the PRINT instruction. You can display data at any screen position, sound a tone when data items are displayed, and clear the screen or a portion of the display row before displaying data.

You can use DISPLAY as either a program statement or a command.

## Cross Reference

DISPLAY USING, GRAPHICS, MARGINS, PRINT

---

!o!   print-list--The print-list consists of one or more print-items
      (items to be displayed on the screen) separated by
      print-separators. See PRINT for an explanation of the
      print-items and print-separators that make up a print-list.

In High-Resolution and Multicolor Modes, items displayed on the screen are not visible. DISPLAY cannot access the graphics portion of the screen in the Split-Screen Modes.

## Options

You can enter the following options, separated by a space, in any order.

!o!    AT--The AT option enables you to specify the beginning of the display field. Row and column are relative to the upper-left corner of the screen window defined by the margins. If you do not use the AT option, the display field begins in the far left column of the bottom row of the current screen window. Before a new line is displayed at the bottom of the window, the entire contents of the window (excluding sprites) scroll up one line to make room for the new line. The contents of the top line of the window scroll off the screen and are discarded. If you use the AT option and your print-list includes a TAB function, the TAB location is relative to the beginning of the display field. If you use the AT option and a print-item is too long to fit in the display field, either the extra characters are discarded (if you use the SIZE option) or the print-item is moved to the beginning of the next screen line (if you do not use the SIZE option).

!o!    BEEP--The BEEP option sounds a short tone when the data items are displayed.

!o!    ERASE ALL--The ERASE ALL option places a space character (ASCII code 32) in every character position in the screen window before displaying the data. The graphics portion of the screen is not affected.

!o!    SIZE--The SIZE option is a numeric-expression whose value specifies the number of character positions to be cleared, starting from the beginning of the display field, before the data is displayed. If the numeric-expression is greater than the number of characters remaining in the row (from the beginning of the display field to the right margin), or if you do not use the SIZE option, the display row is cleared from the beginning of the display field to the right margin.

EXAMPLES
|100 DISPLAY AT(5,7):Y

Displays the value of Y at the fifth row, seventh column of the screen. It first clears row 5 from column 7 to the right margin.


|100 DISPLAY ERASE ALL:B

Puts the blank character into all positions within the current screen window before displaying the value of B.

```
|100 DISPLAY AT(R,C) SIZE(FI
ELDLEN)BEEP:X$
```

Displays the value of X$ at row R, column C.  First it beeps and blanks
FIELDLEN characters.

PROGRAM
The following program illustrates a use of DISPLAY.  It enables you to
position blocks at any screen position to draw a figure or design.

Numbers must be entered as two digits (e.g., 1 would be "01", etc.).  Do not
press ENTER; the information is accepted as soon as the keys are pressed.

This example is valid only in Pattern Mode.

```
|100 CALL CLEAR
|110 CALL COLOR(27,5,5)
|120 DISPLAY AT(23,1):"ENTER
 ROW AND COLUMN."
|130 DISPLAY AT (24,1):"ROW:
    COLUMN:"
|140 FOR COUNT=1 TO 2
|150 CALL KEY(0,ROW(COUNT),S
)
|160 IF S =0 THEN 150
|170 DISPLAY AT(24,5+COUNT)S
IZE(1):STR$(ROW(COUNT)-48)
|180 NEXT COUNT
|190 FOR COUNT=1 TO 2
|200 CALL KEY(0,COLUMN(COUNT
),S)
|210 IF S =0 THEN 200
|220 DISPLAY AT(24,16+COUNT)
SIZE(1):STR$(COLUMN(COUNT)-4
8)
|230 NEXT COUNT
|240 ROW1=10*(ROW(1)-48)+ROW
(2)-48
|250 COLUMN1=10*(COLUMN(1)-4
8)+COLUMN(2)-48
|260 DISPLAY AT(ROW1,COLUMN1
)SIZE(1):CHR$(244)
|270 GOTO 130
```
         (Press CLEAR to stop the program.)

# DISPLAY USING

## Format

DISPLAY [option-list:]USING ;§format-string§;[:print-list]
                          ;§ line-number §;

## Purpose

The DISPLAY USING instruction enables you to define specific formats for numbers and strings you display.

You can use DISPLAY USING as either a program statement or a command.

## Cross Reference

DISPLAY, IMAGE, PRINT

---

!o!  format-string--The format-string specifies the display format. The format string is a string expression; if you use a string constant, you must enclose it in quotation marks. See IMAGE for an explanation of format-strings.

!o!  line-number--You can optionally define a format-string in an IMAGE statement, as specified by the line-number.

!o!  option-list--See DISPLAY under "Options" for an explanation of the options AT, BEEP, ERASE ALL, and SIZE.

!o!  print-list--See PRINT for an explanation of the print-list and print options.

The DISPLAY USING instruction is identical to the DISPLAY instruction with the addition of the USING option, except that:

!o!  You cannot use the TAB function.

!o!  You cannot use any print-separator other than a comma (,), except that the print-list can end with a semicolon (;).

EXAMPLES
```
|100 N=23.43
|110 DISPLAY AT(10,4):USING
"##.##":N
```

Displays the value of N at the tenth row and fourth column, with the format "##.##", after first clearing row 10 from column 4 to the right margin.

```
|100 DISPLAY USING "##.##":N
```

Displays the value of N at the 24th row and first column, with the format "##.##".

## DISTANCE subprogram

### Format

**Two Sprites**
    CALL DISTANCE(#sprite-number1,#sprite-number2,numeric-variable)
**A Sprite and a Screen Pixel**
    CALL DISTANCE(#sprite-number,pixel-row,pixel-column,numeric-variable)

### Purpose

The DISTANCE subprogram enables you to ascertain the distance between two sprites or between a sprite and a specified screen pixel.

### Cross Reference

COINC, SPRITE

---

The DISTANCE subprogram returns the square of the distance sought. (Note that this is not the same as the distance specified by the "tolerance" in the COINC subprogram.)

The square of the distance is the sum of the square of the difference between pixel-rows and the square of the difference between pixel-columns. The distance between the two sprites (or the sprite and the screen pixel) is the square root of the number returned.

If the square of the distance is greater then 32767, the number returned is 32767.

### Two Sprites

The distance between two sprites is considered to be the distance between the upper-left corners of the sprites.

> !o!    sprite-numbers--Sprite-number1 and sprite-number2 are numeric expressions whose values specify the numbers of the two sprites as assigned in the SPRITE subprogram.

> !o!    numeric-variable--The number returned to the numeric-variable equals the square of the distance between the two sprites.

A Sprite and a Screen Pixel

The distance between a sprite and a screen pixel is considered to be the
distance between the upper-left corner of the sprite and the specified pixel.

> !o!  sprite-number--Sprite-number is a numeric expression whose value
>       specifies the number of the sprite as assigned in the SPRITE
>       subprogram.
>
> !o!  pixel-row, pixel-column--The pixel-row and pixel-column are
>       numeric expressions whose values specify the position of the
>       screen pixel.
>
> .!o!  numeric-variable--The number returned to the numeric-variable
>       equals the square of the distance between the sprite and the
>       screen pixel.

EXAMPLES
100 CALL DISTANCE(#3,#4,DIS
T)

Sets DIST equal to the square of the distance between the upper-left corners
of sprite #3 and sprite #4.


100 CALL DISTANCE(#4,18,89,
D)

Sets D equal to the square of the distance between the upper-left corner of
sprite #4 and position 18, 89.

DRAW subprogram

Format

CALL DRAW(line-type,pixel-row1,pixel-column1,pixel-row2,pixel-column2
        [,pixel-row3,pixel-column3,pixel-row4,pixel-column4[,...]])


Purpose

The DRAW subprogram enables you to draw or erase lines between specified
pixels.

Cross Reference

DCOLOR, DRAWTO, FILL, GRAPHICS

---

!o!     line-type--The value of the numeric expression line-type
        specifies the action taken by the DRAW subprogram.

        TYPE    ACTION

        1       Draws a line of the foreground-color specified by the
                DCOLOR subprogram. This is accomplished by turning
                on each pixel in the specified line.

        0       Erases a line. This is accomplished by turning off
                each pixel in the specified line.

        -1      Reverses the status of each pixel on the specified
                line. (If a pixel is on, it is turned off; if a
                pixel is off, it is turned on.) This effectively
                reverses the color of the specified line.


!o!     pixel-row, pixel-column--Pixel-row and pixel-column are numeric
        expressions whose values specify the pixels to be connected by
        the line. You must specify at least two pixels (to define the
        beginning and end points of a line).

        In the Split-Screen Modes, the pixel-row and pixel-column are
        relative to the graphics portion of the screen. The upper-left
        corner of the graphics portion of the screen is considered to be
        the intersection of pixel-row 1 and pixel-column 2.

        In High-Resolution Mode, pixel-row must have a value from 1 to
        192. In the Split-Screen Modes, pixel-row must have a value
        from 1 to 128. In both cases, pixel-column must have a value
        from 1 to 256.

You can optionally draw more lines by specifying additional pairs of pixels. The lines are not connected; each line extends from the first pixel of the pair to the second pixel of the pair. You must specify an even number of pixels.

The last pixel you specify becomes the current position used by the DRAWTO subprogram.

DRAW can be used only in High-Resolution Mode and in the graphics portion of the screen in the Split-Screen Modes. An error results if you use DRAW in Pattern, Text, or Multicolor Mode. DRAW cannot access the text portion of the screen in the Split-Screen Modes.

In High-Resolution Mode, and in the graphics portion of the screen in the Split-Screen Modes, the computer divides each pixel-row into 32 groups of 8 pixels each. (This is most obvious when you assign a background color other than cyan or transparent.) The computer can assign 1 foreground color and 1 background color, from among the 16 available colors, to each 8-pixel group.

PROGRAMS
The following program draws a large triangle on the right side of the graphics portion of the screen.

```
|100 CALL GRAPHICS(X)
|110 CALL CLEAR
|120 CALL DRAW(1,19,185,97,1
15)
|130 CALL DRAW(1,19,185,97,2
55)
|140 CALL DRAW(1,97,115,97,2
55)
|150 GOTO 150
```
        (Press CLEAR to stop the program.)

In line 100, the numeric variable X represents the desired graphics mode (3, 4, or 5,).

The next program uses a FOR-NEXT loop to draw a pattern of lines.

```
|100 CALL CLEAR
|110 CALL GRAPHICS(5)
|120 CALL SCREEN(6)
|130 FOR X=1 TO 255 STEP 5
|140 CALL DRAW(1,1,X,128,256-X)
|150 NEXT X
|160 GOTO 160
```
        (Press CLEAR to stop the program.)

DRAWTO subprogram

Format

CALL DRAWTO(line-type,pixel-row,pixel-column
     [,pixel-row2,pixel-column2[,...]])

Purpose

The DRAWTO subprogram enables you to draw or erase lines between the current position and the specified pixels.

Cross Reference

DCOLOR, DRAW, FILL, GRAPHICS

---

!o! line-type--Line-type is a numeric expression whose value specifies the action taken by the DRAWTO subprogram.

TYPE    ACTION

1       Draws a line of the foreground-color specified by the DCOLOR subprogram. This is accomplished by turning on each pixel in the specified line.

0       Erases a line. This is accomplished by turning off each pixel in the specified line.

-1      Reverses the status of each pixel on the specified line. (If a pixel is on, it is turned off; if a pixel is off, it is turned on.) This effectively reverses the color of the specified line.

!o! pixel-row, pixel-column--The line drawn by DRAWTO extends from the pixel in the current position to the pixel specified by the values of the numeric expressions pixel-row and pixel-column, which becomes the new current position.

You can optionally draw more lines by specifying additional sets of pixels. A line is drawn to each specified pixel from the new current position (the previously specified pixel).

In the Split-Screen Modes, the pixel-row and pixel-column are relative to the graphics portion of the screen. The upper-left corner of the graphics portion of the screen is considered to be the intersection of pixel-row 1 and pixel-column 1.

In High-Resolution Mode, pixel-row must have a value from 1 to 192. In the Split-Screen Modes, pixel-row must have a value from 1 to 128. In both cases, pixel-column must have a value from 1 to 256.

The current position is the last pixel specified the last time the DRAW or the DRAWTO subprogram was called. When you enter TI Extended BASIC II, the current position is the intersection of pixel-row 1 and pixel-column 1.

This default current position is restored only when you change graphics mode.

DRAWTO can be used only in High-Resolution Mode and in the graphics portion of the screen in the Split-Screen Modes. An error results if you use DRAWTO in Pattern, Text, or Multicolor Mode. DRAWTO cannot access the text portion of the screen in the Split-Screen Modes.

In High-Resolution Mode, and in the graphics portion of the screen in the Split-Screen Modes, the computer divides each pixel-row into 32 groups of 8 pixels each. (This is most obvious when you assign a background color other than cyan or transparent.) The computer can assign 1 foreground color and 1 background color (from among the 16 available colors), to each 8-pixel group.

## PROGRAM

The following program uses DRAWTO to create a pattern across the top of the graphics portion of the screen.

```
|100 CALL GRAPHICS(X)
|110 CALL CLEAR
|120 A=20::B=20
|130 CALL DRAW(1,A,B,A,B)
|140 FOR X=1 TO 10
|150 B=B+20
|160 CALL DRAWTO(1,A,B)
|170 CALL DRAWTO(1,A+20,B-20
)
|180 CALL DRAWTO(1,A+20,B)
|190 CALL DRAWTO(1,A,B-20)
|200 NEXT X
|210 GOTO 210
        (Press CLEAR to stop the program.)
```

The numeric variable X in line 100 represents the desired graphics mode (3, 4, or 5).

# END

## Format

END

## Purpose

The END statement stops the execution of your program.

## Cross Reference

STOP

---------------------------------------------------------------------

In addition to terminating program execution, END causes the computer to perform the following operations:

!o!    It closes all open files.

!o!    It restores the default character definitions of all characters.

!o!    If the computer is in High-Resolution or Multicolor Mode, it restores the default graphics mode (Pattern) and margin settings (2, 2, 0, 0).

!o!    It restores the default foreground color (black) and background color (transparent) to all characters.

!o!    It restores the default screen color (cyan).

!o!    It deletes all sprites.

!o!    It resets the sprite magnification level to 1.

The graphics colors (see DCOLOR) and current position (see DRAWTO) are not affected.  If the computer is in Pattern or Text Mode or one of the Split-Screen Modes, the graphics mode and margin settings remain unchanged.

An END statement is not necessary to stop your program; the program automatically stops after the highest numbered line is executed.

END can be used interchangeably with the STOP statement, except that you cannot use STOP after a subprogram.

EOF function--End-of-File

## Format

EOF(file-number)

## Type

INTEGER

## Purpose

The EOF function returns a value indicating whether there are records remaining in a specified file.

-----------------------------------------------------------------

!o!  **file-number**--The **file-number** is a numeric expression whose value specifies the number of the file as assigned in its OPEN instruction.

The value returned by the EOF function depends on the current file position. EOF always treats a file as if it were being accessed sequentially, even if it has been opened for relative access.

| VALUE | MEANING |
|-------|---------|
| 0 | Not end-of-file. |
| (+)1 | Logical end-of-file: No records remaining. |
| -1 | Physical end-of-file: No records remaining, and no space available for more records (storage medium full). |

The EOF function cannot be used with an audio cassette.

For more information about using EOF with a particular device, refer to the owner's manual that comes with that device.

## EXAMPLES
100 PRINT EOF(3)

Prints a value according to whether you are at the end of the file opened as #3.

|100 IF EOF(27) |0 THEN 1150

Transfers control to line 1150 if you are at the end of the file opened as #27.

|100 IF EOF(27) THEN 1150

Transfers control to line 1150 if you are at the end of the file opened as #27.

ERR subprogram--Error

Format

CALL ERR(<u>error-code</u>,<u>error-type</u>[,<u>error-severity</u>,[<u>line-number</u>]])

Purpose

The ERR subprogram enables you to analyze the conditions that caused a program error.

ERR is normally called from a subroutine accessed by an ON ERROR statement.

Cross Reference

ON ERROR

--------------------------------------------------------------------------------
The ERR subprogram returns the <u>error-code</u> and <u>error-type</u>, and optionally the <u>error-severity</u> and <u>line-number</u>, of the most recent "uncleared" program error.

An error is "cleared" when another program error occurs or when the program ends. A RETURN statement in a subroutine accessed by an ON ERROR statement also clears the error.

ON ERROR will not trap an error caused by the RUN command.

!o! <u>error-code</u>--ERR returns a two- or three-digit number to the numeric variable <u>error-code</u>. See Appendix W for a list of error codes and the conditions that cause them to be displayed.

An <u>error-code</u> of 130 indicates an input/output (I/O) error.

An <u>error-code</u> of 0 indicates that no error has occurred.

!o! <u>error-type</u>--The <u>error-type</u> is a numeric variable.

When an I/O error occurs, the value returned in <u>error-type</u> is the number (as assigned in an OPEN instruction) of the file in which the error occurred.

A negative <u>error-type</u> indicates that the error occurred during program execution.

An <u>error-type</u> of 0 indicates that no error has occurred.

Options

> !o! <u>error-severity</u>--The value returned to the numeric variable <u>error-severity</u> is always nine.

> !o! <u>line-number</u>--The value returned to the numeric variable <u>line-number</u> is the line number of the program statement that was executing when the error occurred.

## EXAMPLES
|100 CALL ERR(A,B)

Sets A. equal to the <u>error-code</u> and B equal to the <u>error-type</u> of the most recent error.

|100 CALL ERR(W,X,Y,Z)

Sets W equal to the <u>error-code</u>, X equal to the <u>error-type</u>, Y equal to the <u>error-severity</u>, and Z equal to the <u>line-number</u> of the most recent error.

## PROGRAM
The following program illustrates the use of CALL ERR.

```
|100 ON ERROR 130
|110 CALL SCREEN(18)
|120 STOP
|130 CALL ERR(W,X,Y,Z)
|140 PRINT W;X;Y;Z
|150 RETURN NEXT
|RUN
 79 -1  9   110
```

An error is caused in line 110 by an improper screen-color number. Because of line 100, control is transferred to line 130. Line 140 prints the values obtained. The 79 indicates that a bad value was provided, the -1 indicates that the error occurred during program execution, the 9 is the <u>error-severity</u>, and the <u>110</u> indicates that the error occured in line 110.

EXP function--Exponential

Format

EXP(numeric-expression)

Type

REAL

Purpose

The EXP function returns the value of e raised to the power of the value of the numeric-expression.

EXP is the inverse of the LOG function.

Cross Reference

LOG

--------------------------------------------------------------------------------

The value of e is 2.718281828459.

EXAMPLES
|100 Y=EXP(7)

Assigns to Y the value of e raised to the seventh power, which yields 1096.6331584290.

|100 L=EXP(4.394960467)

Assigns to L the value of e raised to the 4.394960467 power, which yields 81.0414268887.

FILL subprogram

Format

CALL FILL(pixel-row,pixel-column)

Purpose

The FILL subprogram enables you to color the area surrounding a specified
pixel.

Cross Reference

DCOLOR, DRAW, DRAWTO, GRAPHICS

--------------------------------------------------------------------------

!o!     pixel-row, pixel-column--Pixel-row and pixel-column are numeric
        expressions whose values specify the pixel that you want to
        surround with a color.

        In the Split-Screen Modes, the pixel-row and pixel-column are
        relative to the graphics portion of the screen.  The upper-left
        corner of the graphics portion of the screen is considered to be
        the intersection of pixel-row 1 and pixel-column 1.

        In High-Resolution Mode, pixel-row must have a value from 1 to
        192.  In the Split-Screen Modes, pixel-row must have a value
        from 1 to 128.  In both cases, pixel-column must have a value
        from 1 to 256.

The color that surrounds the specified pixel is the foreground color specified
by the DCOLOR subprogram.  If you have not called the DCOLOR subprogram, the
default fill color is black.

The area surrounding the specified pixel is filled with the fill color until a
screen edge or a foreground pixel (a pixel that is turned on) is encountered.
If the boundaries of the fill area are overly complex, FILL may stop before
filling the entire area.

The boundaries of the area to be filled can be defined by lines drawn with the
DRAW and/or DRAWTO subprograms.

FILL can be used only in High-Resolution Mode and in the graphics portion of
the screen in the Split-Screen Modes.  An error results if you use FILL in
Pattern, Text, or Multicolor Mode.  FILL cannot access the text portion of the
screen in the Split-Screen Modes.

In High-Resolution Mode, and in the graphics portion of the screen in the
Split-Screen Modes, the computer divides each pixel-row into 32 groups of 8
pixels each.  The computer can assign a foreground color and a background
color (from among the 16 available colors) to each 8-pixel group.  When a
border that intersects one of these 8-pixel groups is encountered as an area
is filled, any pixels in that group that are already turned on are turned off,
and any pixels in that group that are already off are turned on.

## PROGRAM

The following program divides the graphics portion of the screen into four
horizontal columns and uses FILL to color them.

```
|100 CALL CLEAR
|110 CALL GRAPHICS(5)
|120 CALL DRAW(1,48,0,48,256
)
|130 CALL DRAW(1,96,0,96,256
)
|140 CALL DRAW(1,144,0,144,2
56)
|150 CALL DCOLOR(7,8)
|160 CALL FILL(43,1)
|170 CALL DCOLOR(11,8)
|180 CALL FILL(90,1)
|190 CALL DCOLOR(3,8)
|200 CALL FILL(138,1)
|210 CALL DCOLOR(6,8)
|220 CALL FILL(188,1)
|230 GOTO 230
        (Press CLEAR to stop program.)
```

FOR TO

## Format

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

## Purpose

The FOR TO instruction is used with the NEXT instruction to form a FOR-NEXT loop, which you can use to control a repetitive process.

You can use FOR TO as either a program statement or a command.

## Cross Reference

NEXT

---

## FOR-NEXT Loop Execution

When a FOR TO instruction is executed, the initial-value is assigned to the control-variable. The computer executes instructions until it encounters a NEXT instruction (the group of instructions between the FOR TO and NEXT instructions are known as a "FOR-NEXT loop"). However, if the initial-value is greater than the limit (or, if you specify a negative increment, if the initial-value is less than the limit) the FOR-NEXT loop is not executed.

When the NEXT instruction is encountered, the increment is added to the control-variable; if you do not specify an increment, the control-variable is incremented by 1. Note that if the increment is negative, the value of the control-variable is decreased.

The control-variable in the NEXT instruction must be the same as the control-variable in the FOR TO instruction. The new value of the control-variable is then compared to the limit. If you specify a positive increment (or if you do not specify an increment), the FOR-NEXT loop is repeated if the control-variable is less than or equal to the limit. If you specify a negative increment, the FOR-NEXT loop is repeated if the control-variable is greater than or equal to the limit.

If the condition for repeating the FOR-NEXT loop is met, control passes to the instruction immediately following the FOR TO instruction. If the condition is not met, the FOR-NEXT loop terminates (control passes to the statement immediately following the NEXT statement).

## Specifications

!o!  control-variable--The value of the numeric expression
     control-variable is re-evaluated each time the NEXT instruction
     is executed.  If you change its value while a FOR-NEXT loop is
     executing, you may affect the number of times the loop is
     repeated.  A FOR-NEXT loop executes much faster if the
     control-variable has been declared as an INTEGER than it does if
     the control-variable is REAL.

     The control-variable cannot be an element of an array.

!o!  initial-value--The initial-value is a numeric expression.

!o!  limit--The value of the numeric expression limit is not
     re-evaluated during the execution of a FOR-NEXT loop.  If you
     change its value while a FOR-NEXT loop is executing, you do not
     affect the number of times the loop is repeated.

!o!  increment--The value of the optional numeric expression
     increment is not re-evaluated during the execution of a FOR-NEXT
     loop.  If you change its value while a FOR-NEXT loop is
     executing, you do not affect the number of times the loop is
     repeated.  The increment cannot be zero.

## Nested FOR-NEXT Loops

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

!o!  Each FOR TO instruction must be paired with a NEXT instruction.

!o!  Each nested loop must use a different control-variable.

!o!  If a FOR-NEXT loop contains any portion of another FOR-NEXT
     loop, it must contain all of that FOR-NEXT loop.  If a FOR-NEXT
     loop contains only part of another FOR-NEXT loop, an error
     occurs, and the message NEXT without FOR is displayed.  If the
     FOR-NEXT loop is part of a program, the computer also displays
     the line-number where the error occurred.

## FOR TO as a Program Statement

After you enter the RUN command, but before your program is actually run, the
computer verifies that you have equal numbers of FOR TO and NEXT statements.
If the numbers are not equal, the message FOR-NEXT nesting is displayed and
the program is not run.

You can exit a FOR-NEXT loop by using a GOTO, ON GOTO, or IF THEN statement.
If you use one of these statements to enter a loop, you could cause an error
or create an infinite loop.

A FOR TO statement cannot be part of an IF THEN statement.

FOR TO as a Command

If you use FOR TO as a command, it must be part of a multiple-statement line. A NEXT instruction must also be part of the same line.

After you press ENTER to execute the command, but before the command is actually executed, the computer verifies that you have equal numbers of FOR TO and NEXT instructions. If the numbers are not equal, the message FOR-NEXT nesting is displayed and the command is not executed.

EXAMPLES
```
|100 FOR A=1 TO 5 STEP 2
|110 PRINT A
|120 NEXT A
```

Executes the statements between this 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.

```
|100 FOR J=7 TO -5 STEP -.5
|110 PRINT J
|120 NEXT J
```

Executes the statements between this 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.

PROGRAM
The following program illustrates a use of the FOR-TO-STEP statement. There are three FOR-NEXT loops, with control-variables of CHAR, ROW, and COLUMN.

```
|100 CALL CLEAR
|110 D=0
|120 FOR CHAR=33 TO 63 STEP
30
|130 FOR ROW=1+D TO 21+D STE
P 4
|140 FOR COLUMN=1+D TO 29+D
STEP 4
|150 CALL VCHAR(ROW,COLUMN,C
HAR)
|160 NEXT COLUMN
|170 NEXT ROW
|180 D=2
|190 NEXT CHAR
|200 GOTO 200
```
        (Press CLEAR to stop the program.)

FREESPACE function

## Format

FREESPACE(numeric-expression)

## Type

REAL

## Purpose

The FREESPACE function returns a number representing, in bytes, the amount of memory space available for Extended BASIC II programs and data.

-----------------------------------------------------------------------

!o!   numeric-expression--The value of the numeric-expression must be zero.  Other values are reserved for possible future use.

Garbage Collection

Before FREESPACE returns a value, the computer executes an activity called "garbage collection."

!o!   All "inactive" strings are deleted.  Strings become inactive when they are not associated with a variable.  A string may be created by the computer for its internal use; it becomes inactive when it is no longer needed.

!o!   All "active" strings (strings that are still associated with variables) are moved to a contiguous area at the low end of memory.  This leaves all the available memory in one large, contiguous block.

The computer occasionally performs garbage collection by itself, when no memory is available because of an excessive number and size of inactive strings.

EXAMPLE
PRINT FREESPACE(0)

Prints a value that indicates the amount of available memory.

GCHAR subprogram--Get Character

Format

Pattern, Text, and Multicolor Modes
        CALL GCHAR(row,column,numeric-variable)
High-Resolution and Split-Screen Modes
        CALL GCHAR(pixel-row,pixel-column,numeric-variable)

Purpose

The GCHAR subprogram enables you to ascertain the character code of a character on the screen, the status of a screen pixel, or the color of a block.

Cross Reference

GRAPHICS, HCHAR, VCHAR

-----------------------------------------------------------------------

The meaning of the value returned to the specified numeric-variable varies according to the graphics mode.

Pattern and Text Modes

!o!     row, column--Row and column are numeric expressions whose values specify a character position on the screen.

        The value of row must be greater than or equal to 1 and less than or equal to 24.

        The value of column must be greater than or equal to 1. In Pattern Mode, column must be less than or equal to 32; in Text Mode, column must be less than or equal to 40.

        GCHAR is not affected by margin settings. Row and column are relative to the upper-left corner of the screen, not to the corner of the window defined by the margins.

!o!     numeric-variable--The character code of the character at the specified position is returned to the numeric-variable. See Appendix A for a list of ASCII character codes.

Multicolor Mode

!o!    row, column--Row and column are numeric expressions whose values
       specify a block position on the screen.

       The value of the numeric expression row must be greater than or
       equal to 1 and less than or equal to 48.  The value of the
       numeric expression column must be greater than or equal to 1 and
       less than or equal to 64.

!o!    numeric-variable--The color of the block at the specified
       position is returned to the numeric-variable as a number from 1
       to 16.  See Appendix J for a list of available colors.

High-Resolution and the Split-Screen Modes

!o!    pixel-row, pixel-column--The pixel-row and pixel-column are
       numeric expressions whose values specify a screen pixel position.

       The value of the numeric expression pixel-row must be greater
       than or equal to 1.  In High-Resolution Mode, pixel-row must be
       less than or equal to 192; in the Split-Screen Modes, pixel-row
       must be less than or equal to 128.

       The value of the numeric expression pixel-column must be greater
       than or equal to 1 and less than or equal to 256.

       In the Split-Screen Modes, the pixel-row and pixel-column are
       relative to the graphics portion of the screen.  The upper-left
       corner of the graphics portion of the screen is considered to be
       the intersection of pixel-row 1 and pixel-column 1.

!o!    numeric-variable--The status of the specified screen pixel is
       indicated by the value returned to the numeric-variable.  If the
       pixel is on, the value returned is 1; if the pixel is off, the
       value returned is 0.

In the Split-Screen Modes, GCHAR can access only the graphics portion of the
screen, not the text portion.

EXAMPLES
TI00 CALL GCHAR(12,16,X)

Assigns to X the ASCII code of the character at row 12, column 16 in Pattern and Text Modes. Assigns to X the color of the block at 12, 16 in Multicolor Mode.

|100 CALL GCHAR(R,C,K)

Assigns to K the ASCII code of the character that is in row R, column C in Pattern and Text Modes. Assigns to K the color of the block at R, C in Multicolor Mode.                     the

PROGRAMS
The following program illustrates the use of GCHAR in the Split-Screen Graphics Mode.

|100 CALL CLEAR
|110 CALL GRAPHICS(3)
|120 CALL DRAW(1,0,128,192,1
28)
|130 CALL GCHAR(100,128,X)
|140 CALL GCHAR(100,100,Y)
|150 PRINT X,Y

The variable X is assigned the value 1 because the pixel at row 100, column 128 is on; Y is assigned the value 0 because the pixel at row 100, column 100 is off.

The next program illustrates the use of GCHAR in the Multi-Color Mode.

|100 CALL CLEAR
|110 CALL GRAPHICS(6)
|120 CALL COLOR(16,16,7)
|130 CALL GCHAR(16,16,X)
|140 GOTO 140
        (Press CLEAR to stop the program.)

In line 130, CALL GCHAR returns to X a value of 7 because that is the color of the block located at row 16, column 16.

GOSUB--Go to a Subroutine

Format

;§GOSUB §; line-number
;§GO SUB§;

Purpose

The GOSUB statement transfers program control to the specified subroutine.

A subroutine frequently is used to perform a specific operation several times in the same program.

Cross Reference

ON GOSUB, RETURN

------------------------------------------------------------------------
!o!    line-number--The line-number is a numeric expression whose value
       specifies the program statement at which the subroutine begins.

Use a RETURN statement to return program control to the statement immediately following the GOSUB statement that called the subroutine.

To avoid unexpected results, it is recommended that you exercise care if you use GOSUB to transfer control to or from a subprogram or into a FOR-NEXT loop.

Subroutines may be recursive (self-referencing). To avoid constructing infinite loops, it is recommended that you exercise care when using recursive subroutines.

Nested Subroutines

Subroutines may be "nested"; that is, within a subroutine you can use GOSUB to transfer control to another subroutine. Because RETURN restores program control to the statement immediately following the most recently executed GOSUB, it is important to exercise care when using nested subroutines.

For example, you might use GOSUB in your main program to transfer control to a subroutine, which in turn includes a GOSUB statement that transfers control to another subroutine. When the computer encounters a RETURN in the second subroutine it transfers program control back to the statement immediately following the GOSUB in the first subroutine. Then, when a RETURN is encountered in the first subroutine, program control returns to the statement following the GOSUB in your main program.

## EXAMPLE
```
|100 GOSUB 200
```

Transfers control to statement 200. That statement and the ones up to RETURN are executed and then control returns to the statement after the calling statement.

## PROGRAM
The following program illustrates a use of GOSUB. The subroutine at line 260 figures the factorial of the value of NUMB. The whole program figures the solution to the equation

$$NUMB = X!/(Y! * (X-Y)!)$$

where the exclamation point means factorial. This formula is used to figure certain probabilities. For instance, if you enter X as 52 and Y as 5, you'll find that the number of possible five-card poker hands is 2,598,960. Both numbers entered must be positive integers less than or equal to 69.

```
|100 CALL CLEAR
|110 INPUT "ENTER X AND Y: "
:X,Y
|120 IF X Y THEN 110
|130 IF X|69 OR Y|69 THEN 11
0
|140 IF X 0 THEN PRINT "NEGA
TIVE"::GOTO 110 ELSE NUMB=X
|150 GOSUB 260
|160 NUMERATOR=NUMB
|170 IF Y 0 THEN PRINT "NEGA
TIVE"::GOTO 110 ELSE NUMB=Y
|180 GOSUB 260
|190 DENOMINATOR=NUMB
|200 NUMB=X-Y
|210 GOSUB 260
|220 DENOMINATOR=DENOMINATOR
*NUMB
|230 NUMB=NUMERATOR/DENOMINA
TOR
|240 PRINT "NUMBER IS";NUMB
|250 STOP
|260 REM CALCULATE FACTORIAL
|270 IF NUMB 2 THEN NUMB=1::
GOTO 320
|280 MULT=NUMB-1
|290 NUMB=NUMB*MULT
|300 MULT=MULT-1
|310 IF MULT|1 THEN 290
|320 RETURN
```

GOTO

Format

;§GOTO §; line-number
;§GO TO§;

Purpose

The GOTO statement unconditionally transfers program control to the specified program statement.

Cross Reference

ON GOTO

------------------------------------------------------------------------

!o!   line-number--The line-number is a numeric expression whose value specifies the program statement to which unconditional program control is transferred.

To avoid unexpected results, it is recommended that you exercise care if you use GOTO to transfer control to or from a subroutine or a subprogram or into a FOR-NEXT loop.

PROGRAM

The following program shows the use of GOTO in line 160. Anytime that line is reached, the program executes line 130 next and proceeds from that new point.

```
|100 REM ADD 1 THROUGH 100
|110 ANSWER=0
|120 NUMB=1
|130 ANSWER=ANSWER+NUMB
|140 NUMB=NUMB+1
|150 IF NUMB|100 THEN 170
|160 GOTO 130
|170 PRINT "THE ANSWER IS";A
NSWER
|RUN
THE ANSWER IS 5050
```

GRAPHICS subprogram

Format

CALL GRAPHICS(graphics-mode)

Purpose

The GRAPHICS subprogram enables you to select the graphics-mode that offers you the combination of text and graphics capabilities that best suits the particular needs of your program.

Cross Reference

CHAR, COLOR, DCOLOR, DRAW, DRAWTO, FILL, MARGINS, SCREEN

-----------------------------------------------------------------------

!o! graphics-mode--Graphics-mode is a numeric expression whose value
    is from 1 to 6, specifying one of the six graphics modes
    available in TI Extended BASIC II.

| NUMBER | MODE |
|--------|------|
| 1 | Pattern |
| 2 | Text |
| 3 | Split-Screen: Text-High |
| 4 | Split-Screen: Text-Low |
| 5 | High-Resolution |
| 6 | Multicolor |

When you enter Extended BASIC II, the computer is in Pattern Mode.

Whenever you use the CALL GRAPHICS subprogram, the computer does the following:

!o! Clears the entire screen.

!o! Restores the default character definitions of all characters.

!o! Restores the default foreground color (black) and background
    color (transparent) to all characters.

!o!     Restores the default graphics foreground color (black) and
        background color (transparent).

.!o!    Restores the default screen color (cyan).

!o!     Deletes all sprites.

!o!     Resets the sprite magnification level to 1.

!o!     Restores the default screen margins (2, 2, 0, 0).

!o!     Restores the default current position (pixel-row 1, pixel-column
        1).

!o!     Turns off all sound.


## Pattern Mode

In Pattern Mode, the screen is considered to be a grid 24 characters high and
32 characters wide.  Each character is 8 pixels high and 8 pixels wide.  The
256 available characters are divided into 32 sets of 8 characters each.  You
can use the COLOR subprogram to assign a foreground and a background color,
from among the 16 available colors, to each character set.

In Pattern Mode, you have access to sprites.

The DCOLOR subprogram has no effect in Pattern Mode.  If you use a DRAW,
DRAWTO, or FILL subprogram, the error message Graphics mode error in
line-number is displayed.

## Text Mode

In Text Mode, the screen is considered to be a grid 24 characters high and 40
characters wide.  Each character is 8 pixels high and 6 pixels wide.  (Note
that a character in Text Mode is two pixels narrower than a character in any
other graphics mode.)

You can use the SCREEN subprogram to assign one foreground and one background
color from among the 16 available colors.  The colors you select are assigned
to all 256 characters.

In Text Mode, you do not have access to sprites (the SPRITE subprogram has no
effect in Text Mode).  Using the COLOR subprogram to assign colors to sprites
has no effect.

The DCOLOR subprogram has no effect in Text Mode.  If you use a DRAW, DRAWTO,
or FILL subprogram, the error message Graphics mode error in line-number is
displayed.

The Split-Screen Modes

The Split-Screen Modes divide the screen into two portions. The text portion fills one-third of the screen, and the graphics portion fills the remaining two-thirds of the screen.

In Text-High Mode, the text portion is the upper third of the screen; in Text-Low Mode, the text portion is the lower third of the screen.

In the Split-Screen Modes you have access to sprites, which may appear in either portion of the screen.

Text Portion--The text portion is considered to be a grid 8 characters high and 32 characters wide. Each character is 8 pixels high and 8 pixels wide. You can use the COLOR subprogram to assign a foreground and a background color, from among the 16 available colors, to each character.

The DRAW, DRAWTO, FILL, GCHAR, HCHAR, and VCHAR subprograms cannot access the text portion.

Graphics Portion--The graphics portion is considered to be a grid 128 pixels high and 256 pixels wide.

You can use the DCOLOR subprogram to assign colors to the graphics you display.

The ACCEPT, DISPLAY, DISPLAY USING, INPUT, LINPUT, PRINT, and PRINT USING instructions cannot access the graphics portion.

High-Resolution Mode

In High-Resolution Mode, the screen is considered to be a grid 192 pixels high and 256 pixels wide.

You can use the DCOLOR subprogram to assign colors to the graphics you display. Use the COLOR subprogram only to assign colors to sprites; any other use of the COLOR subprogram causes an error.

In High-Resolution Mode, you have access to sprites.

Data items "displayed" by the ACCEPT, DISPLAY, DISPLAY USING, INPUT, LINPUT, PRINT, and PRINT USING instructions are not visible in High-Resolution Mode.

When a program running in High-Resolution Mode stops running, the computer returns to Pattern Mode.

## Multicolor Mode

In Multicolor Mode, the screen is considered to be a grid 48 blocks high and 64 blocks wide. Each block is 4 pixels high and 4 pixels wide. You can use the COLOR subprogram to assign a color, from among the 16 available colors, to each block.

In Multicolor Mode you have access to sprites.

The DCOLOR, HCHAR, and VCHAR subprograms have no effect in Multicolor Mode. An error occurs if you use the DRAW, DRAWTO, or FILL subprogram. Data "displayed" by the ACCEPT, DISPLAY, DISPLAY USING, INPUT, LINPUT, PRINT, and PRINT USING instructions is not visible.

When a program running in Multicolor Mode stops running, the computer returns to Pattern Mode.

## A Note on High Resolution Graphics

In High-Resolution Mode, and in the graphics portion of the screen in the Split-Screen Modes, the computer divides each pixel-row into 32 groups of 8 pixels. The computer can assign a foreground color and a background color (from among the 16 available colors) to each 8-pixel group.

## EXAMPLES
 CALL GRAPHICS(3)

As a command, changes the current Graphics Mode to Split-Screen, with the text on the upper third of the screen.

|100 CALL GRAPHICS(5)

As a statement, changes the current Graphics Mode to High-Resolution during program execution until execution stops or until another statement changes the Graphics Mode to something else.

HCHAR subprogram--Horizontal Character

## Format

CALL HCHAR(row,column,character-code[number-of-repetitions])

## Purpose

The HCHAR subprogram enables you to place a character on the screen and repeat it horizontally.

## Cross Reference

DCOLOR, GCHAR, GRAPHICS, VCHAR

---

HCHAR has no effect in Multicolor Mode, and cannot access the text portion of the screen in the Split-Screen Modes.

!o!  row, column--Row and column are numeric expressions whose values specify the position on the screen where the character is displayed.

The value of row must be greater than or equal to 1. In Pattern, Text, and High-Resolution Modes, row must be less than or equal to 24; in the Split-Screen Modes, row must be less than or equal to 16.

The value of column must be greater than or equal to 1. In Pattern, High-Resolution, and both Split-Screen Modes, column must be less than or equal to 32; in Text Mode, column must be less than or equal to 40.

HCHAR is not affected by margin settings in Pattern and Text Modes. Row and column are relative to the upper left corner of the screen, not to the corner of the window defined by the margins.

In the Split-Screen Modes, the row and column are relative to the graphics portion of the screen. The upper-left corner of the graphics portion of the screen is considered to be the intersection of row 1 and column 1.

!o!  character-code--Character-code is a numeric expression with a value from 0-255, specifying the code of the character. See Appendix A for a list of ASCII character codes.

!o!    number-of-repetitions--The optional number-of-repetitions is a
       numeric expression whose value specifies the number of times the
       character is repeated horizontally.  If the repetitions extend
       past the end of a row they continue from the first character of
       the next row.  If the repetitions extend past the end of the
       last row they continue from the first character of the first row.

If you use HCHAR to display a character on the screen, and then later use
CHAR, COLOR, or DCOLOR to change the foreground and background colors of that
character, the result depends on the graphics mode.

!o!    In Pattern and Text Modes, the displayed character changes to
       the newly specified pattern and/or color(s).

!o!    In High-Resolution Mode and in the Split-Screen Modes, the
       displayed character remains unchanged.

EXAMPLES
|100 CALL HCHAR(12,16,33)

Places character 33 (an exclamation point) in row 12, column 16.  Remember
that in the Split-Screen Modes, HCHAR positions the output in the graphics
portion of the screen.

|100 CALL HCHAR(1,1,ASC("!")
,768)

Places an exclamation point in row 1, column 1, and repeats it 768 times,
which fills the screen in Pattern Mode.

|100 CALL HCHAR(R,C,K,T)

Places the character with an ASCII code specified by the value of K in row R,
column C and repeats it T times.

Page 83

IF THEN ;§ELSE§;

Format

IF ;§relational-expression§; THEN ;§line-number1§;[ ELSE ;§line-number2§;]
    ;§numeric-expression§;        ;§statement1§;        ;§statement2§;

Purpose

The IF THEN statement enables you to transfer program control to a specified program statement, or to execute a statement or series of statements, based on the status of a condition you specify.

---

The condition tested by the IF THEN statement can be either a relational-expression or a numeric-expression.

!o!     relational-expression--A relational-expression is "true" if it accurately describes the relationship between the variables it references; otherwise, it is "false."

!o!     numeric-expression--A numeric-expression is "false" if it has a value of zero; otherwise, it is "true."

The action specified following THEN or ELSE can be either a line-number or a statement.

!o!     line-number--If the conditional requirement is met and you specify a line-number, program control is transferred to the program statement located at that line-number.

!o!     statement--If the conditional requirement is met and you specify a statement, the specified statement is executed. The statement may be either a single program statement or a series of program statements separated by a double colon (::) statement separator symbol.

If the tested condition is "true," the computer performs the action specified following THEN.

If the tested condition is "false" and you use the ELSE option, the computer performs the action specified following ELSE. Note: A statement separator symbol (::) must not immediately precede ELSE, as this causes a syntax error.

If the tested condition is "false" and you do <u>not</u> use the ELSE option, there are three possibilities.

!o!    If THEN is followed by a <u>statement</u>, program execution proceeds with the next program line.

!o!    If THEN is followed by a <u>line-number</u> <u>only</u>, program execution proceeds with the next program line.

!o!    If THEN is followed by a <u>line-number</u> and a statement separator, program execution proceeds with the statements after the statement separator. <u>Note:</u> In this case, the statement separator symbol functions as an implied ELSE.

An IF THEN statement cannot contain a DEF, DIM, FOR, NEXT, OPTION BASE, SUB, or SUBEND instruction.

EXAMPLES

```
|100 IF X|5 THEN GOSUB 300 E
LSE X=X+5
```

If X is greater than 5, then GOSUB 300 is executed. When the subroutine is ended, control returns to the line following this line. If X is 5 or less, X is set equal to X+5 and control passes to the next line.

```
|100 IF Q THEN C=C+1::GOTO 5
00 ELSE L=L/C::GOTO 300
```

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

```
|100 IF A$="Y" THEN COUNT=CO
UNT+1::DISPLAY AT(24,1):"HER
E WE GO AGAIN!"::GOTO 300
```

If A$ is not equal to "Y", then control passes to the next line. If A$ is equal to "Y", then COUNT is incremented by 1, a message is displayed, and control is transferred to line 300.

```
|100 IF HOURS =40 THEN PAY=H
OURS*WAGE ELSE PAY=HOURS*WAG
E+.5*WAGE*(HOURS-40)::OT=1
```

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

```
|100 IF A=1 THEN IF B=2 THEN
 C=3 ELSE D=4 ELSE E=5
```

If A is not equal to 1, then E is set equal to 5 and control passes to the next line. If A is equal to 1 and B is not equal to 2, then D is set equal to 4 and control passes to the next line. If A is equal to 1 and B is equal to 2, then C is set equal to 3 and control passes to the next line.

PROGRAM

The following program illustrates a use of IF-THEN-ELSE.  It accepts up to
1000 numbers and then prints them in order from smallest to largest.

```
|100 CALL CLEAR
|110 DIM VALUE(1000)
|120 PRINT "ENTER VALUES TO
BE SORTED.":"ENTER '9999' TO
 END ENTRY."
|130 FOR COUNT=1 TO 1000
|140 INPUT VALUE(COUNT)
|150 IF VALUE(COUNT)=9999 TH
EN 170
|160 NEXT.COUNT
|170 COUNT=COUNT-1
|180 PRINT "SORTING."
|190 FOR SORT1=1 TO COUNT-1
|200 FOR SORT2=SORT1+1 TO CO
UNT
|210 IF VALUE(SORT1)|VALUE(S
ORT2)THEN TEMP=VALUE(SORT1):
:VALUE(SORT1)=VALUE(SORT2)::
VALUE(SORT2)=TEMP
|220 NEXT SORT2
|230 NEXT SORT1
|240 FOR SORTED=1 TO COUNT
|250 PRINT VALUE(SORTED)
|260 NEXT SORTED
```

*space*

IMAGE

Format

IMAGE format-string

Purpose

The IMAGE statement enables you to specify the format in which numbers or strings are printed or displayed by a PRINT USING or DISPLAY USING statement.

Cross Reference

DISPLAY USING, PRINT USING

--------------------------------------------------------------------------

!o!     format-string--The format-string is a string constant.

        A format-string containing a quotation mark or leading or
        trailing spaces must be enclosed in quotation marks.  A
        format-string included in a PRINT USING or DISPLAY USING
        statement (rather than as part of an IMAGE statement) must be
        enclosed in quotation marks.

        Any character can be part of a format-string.  Certain
        combinations of characters are interpreted as format-fields, as
        described below.

An IMAGE statement is not executed.

An IMAGE statement cannot be part of a multiple-statement line.


Format-Fields

A format-string can consist of one or more format-fields, each specifying the
format of one print-item.  Format-fields can be separated by any character
except a decimal point or a pound sign.

A format-field may consist of the following characters:

    !o!     A pound sign (#) is replaced by a character from a print-item in
            the print-list of a PRINT USING or DISPLAY USING instruction.
            Allow one pound sign for each digit or character; allow one
            pound sign for the minus sign if necessary.  If you do not allow
            as many pound signs as are necessary to represent the
            print-item, each pound sign is replaced by an asterisk (*).  If
            you use more pound-signs than are necessary to represent the
            print-item, each extra pound sign is replaced by a space.  Added
            spaces precede a number (which right-justifies the number);
            added spaces follow a string (which left-justifies the string).

```
|100 IMAGE ANSWERS ARE ### A
ND ##.##
|110 PRINT USING 100:A,B
```

Allows printing of two numbers. The first may be from -99 to 999 and the second may be from -9.99 to 99.99. The following show how some sample values will be printed or displayed.

| Values | | Appearance |
|---|---|---|
| -99 | -9.99 | ANSWERS ARE -99 AND -9.99 |
| -7 | -3.459 | ANSWERS ARE  -7 AND -3.46 |
| 0 | 0 | ANSWERS ARE   0 AND   .00 |
| 14.8 | 12.75 | ANSWERS ARE  15 AND 12.75 |
| 795 | 852 | ANSWERS ARE 795 AND ***** |
| -984 | 64.7 | ANSWERS ARE *** AND 64.70 |

```
|300 IMAGE DEAR ####,
|310 PRINT USING 300:X$
```

Allows printing a four-character string. The following show how some sample values will be printed or displayed:

| Values | Appearance |
|---|---|
| JOHN | DEAR JOHN, |
| TOM | DEAR TOM , |
| RALPH | DEAR ****, |

*hyph*

## PROGRAMS

The following program illustrates a use of IMAGE. It reads and prints seven numbers and their total.

```
|100 CALL CLEAR
|110 IMAGE $####.##
|120 IMAGE " ####.##"
|130 DATA 233.45,-147.95,8.4
,37.263,-51.299,85.2,464
|140 TOTAL=0
|150 FOR A=1 TO 7
|160 READ AMOUNT
|170 TOTAL=TOTAL+AMOUNT
|180 IF A=1 THEN PRINT USING
 110:AMOUNT ELSE PRINT USING
 120:AMOUNT
|190 NEXT A
|200 PRINT " --------"
|210 PRINT USING "$####.##":
TOTAL
|RUN
```

```
$ 233.45
 -147.95
    8.40
   37.26
  -51.30
   85.20
  464.00
--------
$ 629.06
```

Lines 110 and 120 set up the images.  They are the same except for the dollar sign in line 110.  To keep the blank space where the dollar sign was, the format-string in line 120 is enclosed in quotation marks.

Line 180 prints the values using the IMAGE statements.

Line 210 shows that the format can be put directly in the PRINT USING statement.

The amounts are printed with the decimal points aligned.

The following program shows the effect of using more values in the PRINT USING statement than there are images in the IMAGE statement.

```
|100 IMAGE ###.##,###.#
|110 PRINT USING 100:50.34,5
0.34,37.26,37.26
|RUN
 50.34, 50.3
 37.26, 37.3
```

INIT subprogram--Initialize

Format

CALL INIT[(numeric-expression)]

Purpose

The INIT subprogram reserves memory space to enable the computer to run assembly-language subprograms.

Cross Reference

LINK, LOAD

---

In addition to allocating memory space, INIT removes any assembly-language subprograms that were previously loaded into memory.

!o!     numeric-expression--The value of the optional numeric-expression specifies the number of bytes of memory you want to reserve for assembly-language subprograms.

If you do not enter a numeric-expression, the computer reserves 8K (8192) bytes of memory.

If the value of the numeric-expression is 0, the computer reserves no memory for assembly-language subprograms and releases all memory previously allocated.

The maximum amount of memory space that you can allocate by using INIT is 24,336 bytes.

If you do not call INIT before the first time you use the LOAD subprogram to load an assembly-language subprogram from an external device into memory, the computer reserves 8K bytes of memory.

Although it is not necessary to call INIT in your program, you may wish to do so to remove previously loaded subprograms from memory. Lowering or eliminating the amount of memory reserved for assembly-language subprograms leaves more memory free to be used by Extended BASIC II.

Examples
|CALL INIT

Allocates 8K bytes of memory space.

|CALL INIT(200)

Allocates 200 bytes of memory space.

|CALL INIT(0)

Releases all memory previously allocated.

INPUT

Format

Keyboard Input
        INPUT [input-prompt:]variable-list
File Input
        INPUT #file-number[,REC record-number]

Purpose

The INPUT statement suspends program execution to enable you to enter data from the keyboard. INPUT can be used also to retrieve data from an external device.

Cross Reference

ACCEPT, EOF, LINPUT, OPEN, REC, TERMCHAR

---

!o!   variable-list--The variable-list consists of one or more
      variables separated by commas. Values are assigned to the
      variables in the variable-list in the order they are input. A
      value assigned to a numeric variable must be a number; a value
      assigned to a string variable may be a string or a number.

Variables are assigned values sequentially in the variable-list. A value can be assigned to a variable, and then that variable can be used as a subscript later in the same variable-list.

Input from the Keyboard

If you do not specify a file-number, the program pauses to accept input from the keyboard.

!o!   input-prompt--If you enter an input-prompt, it appears at the
      beginning of the input field, followed immediately by the
      flashing cursor.

      The input-prompt is a string expression; if you use a string
      constant, you must enclose it in quotation marks.

      If you do not enter an input-prompt, a question mark (?) appears
      at the beginning of the input field, followed by a space. The
      flashing cursor appears in the character position following the
      space.

The input field begins in the far left column of the bottom row of the screen window defined by the margins. You can enter up to 157 characters from the keyboard; however, an exceptionally long entry may not be processed correctly by the computer.

The values entered to the variable-list of one INPUT statement must be separated by commas. You must enter the same number of values as there are variables in the variable-list.

A string value entered from the keyboard can optionally be enclosed in quotation marks. However, a string containing a comma, a quotation mark, or leading or trailing spaces must be enclosed in quotation marks. A quotation mark within a string is represented by two adjacent quotation marks.

You normally press ENTER to complete keyboard input; however, you can also use AID, BACK, BEGIN, CLEAR, PROC'D, DOWN ARROW, or UP ARROW. You can use the TERMCHAR function to determine which of these keys was pressed to exit from the previous INPUT, LINPUT, or ACCEPT instruction.

Note that pressing CLEAR during keyboard input normally causes a break in the program. However, if your program includes an ON BREAK NEXT statement, you can use CLEAR to exit from an input field.

The computer sounds a short tone to signal that it is ready to accept keyboard input.

In High-Resolution and Multicolor Modes, neither the input-prompt nor data entered from the keyboard is visible on the screen. INPUT cannot access the graphics portion of the screen in the Split-Screen Modes.

EXAMPLES
|100 INPUT X

Allows the input of a number.


|100 INPUT X$,Y

Allows the input of a string and a number.


|100 INPUT "ENTER TWO NUMBER
S: ":A,B

Displays the prompt ENTER TWO NUMBERS and then allows the entry of two numbers.


|100 INPUT A(J),J

First evaluates the subscript of A and then accepts data into that element of the array A. Then a value is accepted into J.


|100 INPUT J,A(J)

First accepts data into J and then accepts data into the Jth element of the array A.

PROGRAM
The following program illustrates a use of INPUT from the keyboard.

```
|100 CALL CLEAR
|110 INPUT "ENTER YOUR FIRST
 NAME: ":FNAME$
|120 INPUT "ENTER YOUR LAST
NAME: ":LNAME$
|130 INPUT "ENTER A THREE DI
GIT NUMBER: ":DOLLARS
|140 INPUT "ENTER A TWO DIGI
T NUMBER: ":CENTS
|150 IMAGE OF $###.## AND TH
AT IF YOU,
|160 CALL CLEAR
|170 PRINT "DEAR ";FNAME$;",
": : .
|180 PRINT "      THIS IS TO
REMIND YOU"
|190 PRINT "THAT YOU OWE US
THE AMOUNT"
|200 PRINT USING 150:DOLLARS
+CENTS/100
|210 PRINT "IF YOU DO NOT PA
Y US, YOU WILL SOON"
|220 PRINT "RECEIVE A LETTER
 FROM OUR"
|230 PRINT "ATTORNEY, ADDRES
SED TO"
|240 PRINT FNAME$;" ";LNAME$
;"!": :
|250 PRINT TAB(15);"SINCEREL
Y,": : :TAB(15);"I. DUN YOU"
: : : :
|260 GOTO 260
```
         (Press CLEAR to stop the program.)

Lines 110 through 140 allow the person using the program to enter data, as requested with the input-prompts.

Lines 170 through 250 construct a letter based on the input. (Be certain to enter the colons exactly as indicated, because they control line spacing.)

Input from a File

If you include a file-number, input is accepted from the specified device.

 !o!   file-number--The file-number is a numeric expression whose value
       specifies the number of the file as assigned in its OPEN
       instruction.

       If necessary, file-number is rounded to the nearest integer.

 !o!   record-number--If you use the REC option, the record-number is a
       numeric expression whose value specifies the number of the
       record from which you want to input to the variable-list. The
       records in a file are numbered sequentially, starting with
       zero. The REC option can be used only with a file opened for
       RELATIVE access.

       If necessary, record-number is rounded to the nearest integer.

You can accept input only from files opened in INPUT or UPDATE mode. DISPLAY
files must have fewer than 161 characters in each record to be used with an
INPUT statement; however, an exceptionally long record may not be processed
correctly by the computer.

If there are more variables in the variable-list than there are values in the current record, the computer proceeds as follows:

!o! In the case of INTERNAL FIXED records, null strings are assigned to the remaining variables, causing a program error if any of the remaining variables are numeric.

!o! For other records, the computer reads the next record in the file, and uses its values to complete the variable-list.

If there are more values in the current record than are necessary to fill the variable-list, the remaining values are discarded. However, if the variable-list ends with a comma, the computer is placed in an input-pending condition. The remaining values are assigned to the variables in the variable-list of the next INPUT statement unless that statement includes the REC option, in which case the remaining values are discarded.

EXAMPLES
|100 INPUT #1:X$

Puts into X$ the next value available in the file that was opened as #1.

|100 INPUT #23:X,A,LL$

Puts into X, A, and LL$ the next three values from the file that was opened as #23 with data in INTERNAL format.

|100 INPUT #11,REC 44:TAX

Puts into TAX the first value of record number 44 of the file that was opened as #11 with RELATIVE file organization.

|100 INPUT #3:A,B,C,
|110 INPUT #3:X,Y,Z

Puts into A, B, and C the next three values from the file opened as #3. The comma after C creates an pending input condition, and because the INPUT statement in line 110 has no REC clause, the computer assigns to X, Y, and Z data values beginning where the previous INPUT statement stopped.

PROGRAM
The following program illustrates a use of the INPUT statement. It opens a
file on the cassette recorder and writes 5 records on the file. It then goes
back and reads the records and displays them on the screen.

See the HEX-BUS™ Disk Drive/Controller manual for instructions on using
diskettes.

```
|100 OPEN #1:"CS1",SEQUENTIA
L,INTERNAL,OUTPUT,FIXED 64
|110 FOR A=1 TO 5
|120 PRINT #1:"THIS IS RECOR
D",A
|130 NEXT A
|140 CLOSE #1
|150 CALL CLEAR
|160 OPEN #1:"CS1",SEQUENTIA
L,INTERNAL,INPUT,FIXED 64
|170 PRINT
|180 FOR B=1 TO 5
|190 INPUT #1:A$,C
|200 PRINT A$;C
|210 NEXT B
|220 CLOSE #1
|RUN
THIS IS RECORD 1
THIS IS RECORD 2
THIS IS RECORD 3
THIS IS RECORD 4
THIS IS RECORD 5
REWIND CASSETTE TAPE
THEN PRESS ENTER

PRESS CASSETTE RECORD
THEN PRESS ENTER

PRESS CASSETTE STOP
THEN PRESS ENTER

REWIND CASSETTE TAPE
THEN PRESS ENTER

PRESS CASSETTE PLAY
THEN PRESS ENTER

PRESS CASSETTE STOP
THEN PRESS ENTER
```

INT function--Integer

Format

INT(numeric-expression)

Type

Real

Purpose

The INT function returns the largest integer not greater than the value of the numeric-expression.

---

!o!    numeric-expression--If the value of the numeric-expression is an integer, INT returns the value of the numeric-expression itself.  If the numeric-expression is not an integer, INT returns the largest integer not greater than the numeric-expression.

EXAMPLES
|100 PRINT INT(3.4)

Prints 3.

|100 X=INT(3.9)

Sets X equal to 3.

|100 P=INT(3.9999999999)

Sets P equal to 3.

|100 DISPLAY AT(3,7):INT(4.0
)

Displays 4 at the third row, seventh column of the current screen window.

|100 N=INT(-3.9)

Sets N equal to -4.

|100 K=INT(-3.0000001)

Sets K equal to -4.

INTEGER

Format

INTEGER <u>numeric-variable-list</u>
INTEGER ALL

Purpose

The INTEGER instruction enables you to declare the data-type of specified numeric variables as INTEGER.

INTEGER variables are processed faster and require less memory than do REAL variables.

You can use INTEGER as either a program statement or a command.

Cross Reference

DEF, DIM, OPTION BASE, REAL, SUB

----------------------------------------------------------------------

!o!  <u>numeric-variable-list</u>--The <u>numeric-variable-list</u> consists of one or more numeric variables separated by commas. The variables are all assigned the INTEGER data-type. An INTEGER statement with a <u>numeric-variable-list</u> must have a lower line number than any program reference to any variable in that list.

!o!  ALL--If you enter the ALL option, all numeric variables in your program are assigned the INTEGER data-type unless specifically declared as REAL. An INTEGER statement with the ALL option must have a lower line number than any program reference to any numeric variable or array.

An INTEGER ALL command does not affect any variables unless they follow INTEGER ALL on a multiple-statement line.

An INTEGER ALL statement in your main program does not affect the data-type of a numeric variable in a subprogram.

A numeric variable of the INTEGER data-type is a whole number greater than or equal to -32768 and less than or equal to 32767.

If you do not specify the data-type of a numeric variable, it is assigned the REAL data-type (unless your program includes an INTEGER ALL statement).

If the value of an INTEGER variable is changed so that it is no longer a whole number, its value is rounded to the nearest whole number. If a change to an INTEGER variable would place that variable outside the acceptable INTEGER range, a program error occurs, and the value of the variable remains unchanged.

INTEGER statements are evaluated during pre-scan, and are not executed.

You can also declare INTEGER variables by using the data-type option in the DEF, DIM and SUB statements.

EXAMPLES
TINTEGER B

As a command, specifies that the variable B is an integer.

|100 INTEGER ALL

As a statement specifies that all variables within the program are integers.

|100 INTEGER M(20)

Reserves space in memory for 21 integer elements of the array M.

JOYST subprogram--Joystick

## Format

CALL JOYST(key-unit,x,y)

## Purpose

The JOYST subprogram enables you to ascertain the position of either of the Joystick Controllers.

----------------------------------------------------------------

!o!   key-unit--The numeric expression key-unit can have a value of 1 or 2, specifying the joystick you are testing.

!o!   x,y--The position of the specified joystick is returned in the numeric variables x and y as follows:

| POSITION | X | Y |
|---|---|---|
| Center | 0 | 0 |
| Up | 0 | (+)4 |
| Upper Right | (+)4 | (+)4 |
| Right | (+)4 | 0 |
| Lower Right | (+)4 | -4 |
| Down | 0 | -4 |
| Lower Left | -4 | -4 |
| Left | -4 | 0 |
| Upper Left | -4 | (+)4 |

If the specified joystick is not connected to the computer, x and y are both returned as 0.

## EXAMPLE
TIOO CALL JOYST(1,X,Y)

Returns values in X and Y according to the position of joystick number 1.

## PROGRAM

The following program illustrates a use of the JOYST subprogram.  It creates a sprite and then moves it around according to the input from a joystick.

```
|100 CALL CLEAR
|110 CALL SPRITE(#1,33,5,96,
128)
|120 CALL JOYST(1,X,Y)
|130 CALL MOTION(#1,-Y*4,X*4
)
|140 GOTO 120
        (Press CLEAR to stop the program.)
```

KEY subprogram

Format

CALL KEY(key-unit,key,status)

Purpose

The KEY subprogram enables you to transfer one character from the keyboard directly to a program.

KEY can sometimes replace an INPUT statement, especially for the input of a single character.

---

!o!   key-unit--The numeric expression key-unit can have a value from 0 to 5, as explained below.

!o!   key--The character code of the key pressed is returned in the numeric variable key. If no key is pressed, a value of -1 is returned.

See Appendix A for a list of the available characters. See Appendix D for an illustration showing the character codes returned by each key.

!o!   status--The keyboard status is returned in the numeric variable status as explained below.

Because the character represented by the key pressed is not displayed on the screen, the information already on the screen is not disturbed.

Specifying 3, 4, or 5 as the key-unit maps the keyboard to a particular mode of operation. The mode you specify determines which character-codes are returned when certain keys are pressed. The keyboard mode remains in effect until the program ends (either normally or because of an error), stops at a breakpoint, or another keyboard mode is specified.

Specifying 1 or 2 as the key-unit affects only the values returned by that CALL KEY, and does not affect the keyboard mode.

Key-Unit Options

The value you specify for the key-unit determines what portion of the keyboard is active and how the key pressed is interpreted.

| KEY-UNIT | RESULT |
|---|---|
| 0 | Uses either key-unit 3, 4, or 5, depending on which of those key-units was most recently specified in a call to the KEY subprogram.  If you have not previously called KEY, the computer uses key-unit 5. |
| 1 | Only the left side of the keyboard is active. |
| 2 | Only the right side of the keyboard is active. |
| 3 | TI-99/4 Emulation:  Returns a value as if a key had been pressed on the keyboard of a TI-99/4 Home Computer; also returns codes for the left and right brackets ([ and ]) and the backslash ( ) keys.  All alphabetic characters . are returned as upper case; the CAPS key is locked; the control keys return codes 13 and 32; the function keys return codes 1-15, and 32. |
| 4 | Pascal Emulation:  Returns a value as if a key had been pressed while you were running a program written in Pascal.  The control keys return codes 0-32, 176-183, and 187; the function keys return codes 13, 32, 127, 129-140, 142, 143, 184-186, and 188-198. |
| 5 | BASIC Emulation:  Returns normal TI Extended BASIC II values.  The control keys return codes 13, 32, 128-159, 176-183, and 187; the function keys return codes 1-15, 32, 127, 184-186, and 188-198. |

## Status

The value returned as the status can be interpreted as follows:

!o!    -1 means the same key was pressed as was returned the last time KEY was called.

!o!    0 means no key was pressed.

!o!    (+)1 means a different key was pressed than was returned the last time KEY was called.

## EXAMPLE
TIOO CALL KEY(0,K,S)

Returns in K the ASCII code of any key pressed on the keyboard except SHIFT, CTRL, FCTN, and CAPS, and in S a value indicating whether a key was pressed.

PROGRAM

The following program illustrates a use of the KEY subprogram. It creates a sprite and then enables you to move it around ~~according~~ by using the arrow keys (E, S, D, and X) without pressing FCTN. Note that line 130 returns to line 120 if no key has been pressed.

To stop the sprite's movement, press any key (except the arrow keys) on the left side of the keyboard.

```
|100 CALL CLEAR
|110 CALL SPRITE(#1,33,5,96,
128)
|120 CALL KEY(1,K,S)
|130 IF S=0 THEN 120
|140 IF K=5 THEN Y=-4
|150 IF K=0 THEN Y=4
|160 IF K=2 THEN X=-4
|170 IF K=3 THEN X=4
|180 IF K=1 THEN X,Y=0
|190 IF K|5 THEN X,Y=0
|200 CALL MOTION(#1,Y,X)
|210 GOTO 120
        (Press CLEAR to stop the program.)
```

LEN function--Length

**Format**

LEN(<u>string-expression</u>)

**Type**

INTEGER

**Purpose**

The LEN function returns the number of characters in the string specified by
the <u>string-expression</u>.

--------------------------------------------------------------------------------
If the <u>string-expression</u> is a null string, LEN returns a zero.

Remember that a space is a valid character and is considered to be part of the
length of a string.

EXAMPLES
|100 PRINT LEN("ABCDE")

Prints 5.

|100 X=LEN("THIS IS A SENTEN
CE.")

Sets X equal to 19.

|100 DISPLAY LEN("")

Displays 0.

|100 DISPLAY LEN(" ")

Displays 1.

|100 A$="DAVID"
|110 DISPLAY LEN(A$)

Displays 5 when A$ equals DAVID.

LET

## Format

[LET ]variable-list=expression

## Purpose

The LET instruction, often called the "assignment" instruction, enables you to assign values to variables.

You can use LET as either a program statement or a command.

---

!o!     variable-list--The variable-list consists of one or more variables separated by commas.  Do not mix numeric and string variables in the same variable-list.  However, you can include both INTEGER and REAL numeric variables in the same variable-list.

!o!     expression--The value of the expression is assigned to all variables in the variable-list.  If the variable-list contains numeric variables, the expression must be a numeric expression. If the variable-list contains string variables, the expression must be a string expression.

The word LET can optionally be omitted from the instruction.

## EXAMPLES
|100 T=4

Assigns to T the value 4.

|100 X,Y,Z=12.4

Assigns to X, Y, and Z the value 12.4.

|100 A=3 5

Assigns -1 to A because it is true that 3 is less than 5.

|100 B=12 7

Assigns 0 to B because it is not true that 12 is less than 7.

```
|100 L$,D$,B$="B"
```

Assigns to L$, D$, and B$ the string constant "B".


## PROGRAM

The following program illustrates a use of LET.

```
|100 K=1
|110 K,A(K)=3
|120 PRINT K;A(1)
|130 PRINT A(3);A(K)
|RUN
  3  3
  0  0
```

In line 100, the variable K is assigned the value 1.

In line 110, the variable K and the array element A(K) are assigned the value 3. Note that when line 110 is executed, the subscript K is not assigned a new value, but has the same value it had before the line was executed. Therefore, A(K) is an expression equivalent to A(1), referring to the same element of the array.

In line 120, the values of K and A(1) are printed.

When line 130 is executed, K has a value of 3; therefore, A(K) is now an expression equivalent to A(3). Both expressions have a value of 0 (the default value) because no value has been assigned to this element of the array.

LINK subprogram

Format

CALL LINK(subprogram-name[,parameter-list])

Purpose

The LINK subprogram enables you to transfer control from a TI Extended BASIC II program to an assembly-language subprogram.

Cross Reference

INIT, LOAD, SUB

---

!o!    subprogram-name--The subprogram-name is an entry point in an assembly-language subprogram that you have previously loaded into memory with the LOAD subprogram. The subprogram-name is a string expression; if you use a string constant, it must be enclosed in quotation marks.

!o!    parameter-list--The optional parameter-list consists of one or more parameters, separated by commas, that are to be passed to the assembly-language subprogram. The contents of the parameter-list depend on the particular subprogram you are accessing.

The rules for passing parameters to an assembly-language subprogram are the same as the rules for passing parameters to an Extended BASIC II subprogram (see SUB).

EXAMPLE
T100 CALL LINK("START",1,3)

Links the TI Extended BASIC II program to the assembly-language subprogram START, and passes the values 1 and 3 to it.

LINPUT--Line Input

Format

Keyboard Input
       LINPUT [input-prompt:]string-variable
File Input
       LINPUT #file-number[,REC record-number]:string-variable

Purpose

The LINPUT statement suspends program execution to enable you to enter a line
of unedited data from the keyboard.  LINPUT can be used also to retrieve an
unedited record from an external device.

Cross Reference

ACCEPT, EOF, INPUT, OPEN, TERMCHAR

----------------------------------------------------------------------------
       !o!     string-variable--LINPUT assigns an entire line, a file record,
               or the remaining portion of a file record (if there is a pending
               input condition) to the string-variable.

       !o!     input-prompt, file-number, record-number--See INPUT for an
               explanation of keyboard and file input and input options.

No editing is performed on the input data.  All characters (including commas,
quotation marks, colons, semicolons, and leading and trailing spaces) are
assigned to the string-variable as they are encountered.

The maximum value that can be input from the keyboard is 255 characters.

LINPUT is frequently used instead of INPUT when the input data may include a
comma.  (A comma is not accepted as input by the INPUT statement, except as
part of a string enclosed in quotation marks.)

To use LINPUT for file input the file must be in DISPLAY format.

You normally press ENTER to complete keyboard input; however, you can also use
AID, BACK, BEGIN, CLEAR, PROC'D, DOWN ARROW, or UP ARROW.  You can use the
TERMCHAR function to determine which of these keys was pressed to exit from
the previous ACCEPT, INPUT, or LINPUT instruction.

Note that pressing CLEAR during keyboard input normally causes a break in the
program.  However, if your program includes an ON BREAK NEXT statement, you
can use CLEAR to exit from an input field.

## EXAMPLES
|100 LINPUT L$

Assigns to L$ anything typed before <u>ENTER</u> is pressed.

|100 LINPUT "NAME: ":NM$

Displays NAME: and assigns to NM$ anything typed before <u>ENTER</u> is pressed.

|100 LINPUT #1,REC M:L$(M)

Assigns to L$(M) the value that was in record M of the file that was opened as #1 with RELATIVE DISPLAY file organization.

## PROGRAM
The following program illustrates the use of LINPUT.  It reads a previously existing file and displays only the lines that contain the word "THE."

```
|100 OPEN #1:"DSK1.TEXT1",IN
PUT,FIXED 80,DISPLAY
|110 IF EOF(1) THEN CLOSE #1
 :: STOP
|120 LINPUT #1:A$
|130 X=POS(A$,"THE",1)
|140 IF X |0 THEN PRINT A$
|150 GOTO 110
```

## LIST

### Format

List to the Screen
     LIST [line-number-range]
List to a File (or Device)
     LIST "file-specification"[:line-number-range]

### Purpose

The LIST command displays the program (or a portion of it) currently in memory.  You can also use LIST to output the program listing to an external device.

------------------------------------------------------------------------

!o!     line-number-range--The optional line-number-range specifies the portion of the program to be listed.  If you do not enter a line-number-range, the entire program is listed.  The program lines are always listed in ascending order.

!o!     file-specification--If you enter a file-specification, the program listing is output to the specified file or device.  The file-specification, a string constant, must be enclosed in quotation marks.  For more information see ~~Section X,~~ "File Specifications." *beginning on page X.*

The program listing is output as a SEQUENTIAL file in DISPLAY format with VARIABLE records (see OPEN); the file-specification option can be used only with devices that accept these options.  For more information about listing a program on a particular device, refer to the owner's manual that comes with that device.  If you do not enter a file-specification, the program listing is displayed on the screen.

You can stop the listing at any time by pressing CLEAR (FCTN 4).  Pressing any other key (except SHIFT, FCTN, or CTRL) causes the listing to pause until you press a key again.

The LIST command ~~will~~ only work with peripherals that support DISPLAY/VARIABLE type records.

### The Line-Number-Range

A line-number-range can consist of a single line number, a single line number followed by a hyphen, a single line number preceded by a hyphen, or a range of line numbers.

| COMMAND | LINES LISTED |
|---|---|
| LIST | All lines |
| LIST X | Line number X only |
| LIST X- | Lines from number X to the highest line number, inclusive |
| LIST -X | Lines from the lowest line number to line number X, inclusive |
| LIST X-Y or LIST X Y | All lines from line number X to line number Y, inclusive |

If the line-number-range does not include a line number in your program, the following conventions apply:

!o!   If line-number-range is higher than any line number in the program, the highest-numbered program line is listed.

!o!   If line-number-range is lower than any line number in the program, the lowest-numbered program line is listed.

!o!   If line-number-range is between lines in the program, the next higher numbered program line is listed.

EXAMPLES
TLIST

Lists the entire program in memory on the display screen.

|LIST 100

Lists line 100.

|LIST 100-

Lists line 100 and all lines after it.

|LIST -200

Lists all lines up to and including line 200.

|LIST 100-200

Lists all lines from 100 through 200.

Page 115

LOAD subprogram

Format

File Only
      CALL LOAD(file-specification-list)
Data Only
      CALL LOAD(address,byte-list[,"",address,byte-list[,...]])
File and Data
      ;§CALL LOAD(file-specification-list,address,byte-list[,...])§;
      ;§CALL LOAD(address,byte-list,file-specification-list[,...])§;


Purpose

The LOAD subprogram enables you to load assembly-language subprograms into memory.  You can also use LOAD to assign values directly to specified CPU (Central Processing Unit) memory addresses.

You can use the POKEV subprogram to assign values to VDP (Video Display Processor) memory.

Cross Reference

INIT, LINK, PEEK, PEEKV, POKEV, VALHEX

-------------------------------------------------------------------------

To load an assembly-language subprogram, specify a file-specification-list; to assign values to CPU memory, specify an address and a byte-list (an address must always be followed by a byte-list).

You must enter at least one parameter.  The first parameter you specify can be either a file-specification-list or an address.

If you wish to follow an address and byte-list with another address and byte-list, enter a file-specification-list or a null string (two adjacent quotation marks) as a separator.

   !o!   file-specification-list--The optional file-specification-list consists of one or more file-specifications separated by commas (see Section X, "File Specifications").  A file-specification is a string expression; if you use a string constant, you must enclose it in quotation marks.

         Each file-specification names an assembly-language object (program) file to be loaded into memory.  The specified file can include subprogram names, so that the subprograms can be executed by the LINK subprogram.

The object file to be loaded must be in DISPLAY format with FIXED records (see OPEN). For more information about the file options available with a particular device, refer to the owner's manual that comes with that device.

!o!    address--You can optionally load bytes of data to a specified CPU memory address. The address specifies the first address where the data is to be loaded; if the byte-list specifies more than one byte of data, the bytes are assigned to sequential memory addresses starting with the address you specify.

The numeric expression address must have a value from -32768 to 32767 inclusive.

You can specify an address from 0 to 32767 inclusive by specifying the actual address.

You can specify an address from 32768 to 65535 inclusive by subtracting 65536 from the actual address. This will result in a value from -32768 to -1 inclusive.

If you know the hexadecimal value of the address, you can use the VALHEX function to convert it to a decimal numeric expression, eliminating the possible need for calculations.

If necessary, the address is rounded to the nearest integer.

!o!    byte-list--The byte-list consists of one or more bytes of data, separated by commas, that are to be loaded into CPU memory starting with the specified address.

Each byte in the byte-list must be a numeric expression with a value from 0 to 32767. If the value of a byte is greater than 255, it is repeatedly reduced by 256 until it is less than 256. If necessary, a byte is rounded to the nearest integer.

Note that you must use the INIT subprogram to reserve memory space before you use LOAD to load a subprogram.

If you call the LOAD subprogram with invalid parameters or load an object file with absolute (rather than relocatable) addresses, the computer may function erratically or cease to function entirely. If this occurs, turn off the computer, wait several seconds, then turn the computer back on again.

The Loader

LOAD uses a "relocatable linking" loader.

Because it is "relocatable," you cannot use LOAD to specify a memory <u>address</u> at which you want to load a file.  However, the file you are loading may specify an absolute load <u>address</u> if it includes an AORG directive.

Because it is "linking," the object files specified in the <u>file-specification-list</u> can reference each other.


<u>EXAMPLE</u>
|CALL LOAD(VALHEX("849E"),X,0)

Enables you to specify the execution speed.  The numeric variable $X$ can be 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.  Assigning the value 0 to X specifies that execution proceeds at full speed.  Assigning the value 2 to X specifies that the execution of assembly-language programs is slowed to approximately TI-99/4A Home Computer speed.  Experiment with the value of X to slow the execution of TI Extended BASIC II programs to various speeds.  (The greater the value of X, the slower will be the the execution speed.)


|CALL LOAD(VALHEX("84BD"),1)

Turns on the "key-chirp" (a sound emitted whenever a key is pressed).


|CALL LOAD(VALHEX("84BD"),0)

Turns off the key-chirp.


|CALL LOAD("DSK1.PROGRAM")

Loads an assembly-language program named PROGRAM into memory.

LOCATE subprogram

Format

CALL LOCATE(#sprite-number,pixel-row,pixel-column[,...])

Purpose

The LOCATE subprogram enables you to change the location of one or more sprites.

Cross Reference

DELSPRITE, SPRITE

---

!o!  sprite-number--The sprite-number is a numeric expression whose value specifies the number of a sprite as assigned by the SPRITE subprogram.

!o!  pixel-row, pixel-column--The pixel-row and pixel-column are numeric expressions whose values specify the screen pixel location of the pixel at the upper-left corner of the sprite.

LOCATE can cause a sprite that has been deleted with DELSPRITE sprite-number to reappear.

PROGRAM
The following program illustrates the use of the LOCATE subprogram.

```
|100 CALL CLEAR
|110 CALL SPRITE(#1,33,7,1,1
,25,25)
|120 YLOC=INT(RND*150+1)
|130 XLOC=INT(RND*200+1)
|140 FOR DELAY=1 TO 300:
NEXT DELAY
|150 CALL LOCATE(#1,YLOC,XLO
C)
|160 GOTO 120
```
        (Press CLEAR to stop the program.)

Line 110 creates a sprite as a fairly quickly moving red exclamation point.

Line 140 locates the sprite at a location randomly chosen in lines 120 and 130.

Line 150 repeats the process.

Also see the third example of the SPRITE subprogram, on page X.

2171L--Computer 99/8 Reference Guide--Final Draft 09/15/83

## LOG function--Natural Logarithm

**Format**

LOG(<u>numeric-expression</u>)

**Type**

REAL

**Purpose**

The LOG function returns the natural logarithm of the value of the <u>numeric-expression</u>.

LOG is the inverse of the EXP function.

**Cross Reference**

EXP

------------------------------------------------------------------------

!o! numeric-expression--The value of the <u>numeric-expression</u> must be greater than zero.

EXAMPLES
|100 PRINT LOG(3.4)

Prints the natural logarithm of 3.4, which is 1.223775432.

|100 X=LOG(EXP(7.2))

Sets X equal to the natural logarithm of e raised to the 7.2 power, which is 7.2.

|100 S=LOG(SQR(T))

Sets S equal to the natural logarithm of the square root of the value of T.

Page 120

## PROGRAM
The following program returns the logarithm of any positive number in any base.

```
|100 CALL CLEAR
|110 INPUT "BASE:  ":B
|120 IF B =1 THEN 110
|130 INPUT "NUMBER: ":N
|140 IF N =0 THEN 130
|150 LG=LOG(N)/LOG(B)
|160 PRINT "LOG BASE";B;"OF"
;N;"IS";LG
|170 PRINT
|180 GOTO 110
```
(Press CLEAR to stop the program.)

MAGNIFY subprogram

Format

CALL MAGNIFY(numeric-expression)

Purpose

The MAGNIFY subprogram enables you to specify whether all sprites are single- or double-sized and whether they are unmagnified or magnified.

Cross Reference

CHAR, SPRITE

--------------------------------------------------------------------

!o!  numeric-expression--The value of the numeric-expression
     specifies the size and magnification "level" of all sprites.
     (You cannot specify the level of an individual sprite.)

LEVEL            CHARACTERISTICS

  1              Single-sized, unmagnified
  2              Single-sized, magnified
  3              Double-sized, unmagnified
  4              Double-sized, magnified

The screen position of the pixel in the upper-left corner of a sprite is considered to be the position of that sprite. That pixel remains in the same screen position regardless of changes to the magnification level.

When you enter TI Extended BASIC II, sprites are single-sized and unmagnified (level 1). When your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode, the sprite magnification level is restored to 1.

Single-Sized Sprites

A single-sized sprite is defined only by the character you specify when the sprite is created.

Double-Sized Sprites

A double-sized sprite is defined by four consecutive characters, including the character that you specify when the sprite is created.

If the number of the character you specify is a multiple of 4, that character is the first of the four characters that comprise the sprite's definition. If the character number is not a multiple of 4, the next lower character that is a multiple of four is the first character of the sprite.

The first of the four characters defines the upper-left quarter of the sprite, the second character defines the lower-left quarter of the sprite, the third defines the upper-right quarter of the sprite, and the last of the four characters defines the lower-right quarter of the sprite.

Unmagnified Sprites

An unmagnified sprite occupies only the number of characters on the screen specified by the characters that define it.

A single-sized unmagnified sprite occupies 1 character position on the screen; a double-sized unmagnified sprite occupies 4 character positions.

Magnified Sprites

A magnified sprite expands to twice the height and twice the width of an unmagnified sprite. The expansion occurs down and to the right; the pixel in the upper-left corner of the sprite remains in the same screen position.

A magnified sprite has 4 times the area of an unmagnified sprite. When you magnify a sprite, each pixel of the unmagnified sprite expands to 4 pixels of the magnified sprite.

A single-sized magnified sprite occupies 4 character positions on the screen; a double-sized magnified sprite occupies 16 character positions.

PROGRAM

The following program illustrates a use of the MAGNIFY subprogram.

A little figure (single-sized, unmagnified) appears near the center of the screen. In a moment, it becomes twice as big (single-sized, magnified), covering four character positions. In another moment, it is replaced by the upper-left corner of a larger figure (single-sized, magnified), still covering four character positions. Then the full figure appears (double-sized, magnified), covering sixteen character positions. Finally it is reduced in size to four character positions (double-sized, unmagnified).

```
|100 CALL CLEAR
|110 CALL CHAR(248,"1898FF3D
3C3CE404")
|120 CALL SPRITE(#1,248,5,92
,124)
|130 GOSUB 230
|140 CALL MAGNIFY(2)
|150 GOSUB 230
|160 CALL CHAR(248,"0103C341
7F3F07070707077E7C40000080C0
C080FCFEE2E3E0E0E06060606070
")
|170 GOSUB 230
|180 CALL MAGNIFY(4)
|190 GOSUB 230
|200 CALL MAGNIFY(3)
|210 GOSUB 230
|220 STOP
|230 REM DELAY
|240 FOR DELAY=1 TO 500
|250 NEXT DELAY
|260 RETURN
```

Line 110 defines character 248.

Line 120 sets up a sprite using character 248.  By default the magnification factor is 1.

Line 140 changes the magnification factor to 2.

Line 160 redefines character 248.  Because the definition is 64 characters long, it also defines characters 249, 250, and 251.

Line 180 changes the magnification factor to 4.

Line 200 changes the magnification factor to 3.

MARGINS subprogram

Format

CALL MARGINS(left,right,top,bottom)

Purpose

The MARGINS subprogram enables you to define screen margins.  The margins you specify define a screen window that affects the operation of several instructions.

Cross Reference

ACCEPT, CLEAR, DISPLAY, DISPLAY USING, GRAPHICS, INPUT, LINPUT, PRINT, PRINT USING

--------------------------------------------------------------------------------

!o!    left, right, top, bottom--Left, right, top, and bottom are numeric expressions whose values specify the margin indention from the four edges of the screen.

The margins cannot "overlap"; that is, the position of the top margin must be higher on the screen than the bottom margin, and the position of the left margin must be farther left on the screen than the right margin.

When creating a screen window, you must leave the window large enough to allow entry of a command.

The valid range for margin indention varies according to the graphics mode. Acceptable values for the sum of opposing margins in each mode are as follows:

| MODE | TOP&BOTTOM | LEFT&RIGHT |
|------|-----------|-----------|
| Pattern | 0-23 | 0-31 |
| Text | 0-23 | 0-39 |
| Split-Screen | 0-7 | 0-31 |

The upper-left corner of the window defined by the margins is considered to be the intersection of row 1 and column 1 by the ACCEPT, DISPLAY, and DISPLAY USING instructions that use the AT option.

The lower-left corner of the window is considered to be the beginning of the input line by the ACCEPT, INPUT, and LINPUT instructions.

The lower-left corner of the window is considered to be the beginning of the print line by the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions.

When the ACCEPT, INPUT, LINPUT, PRINT, or PRINT USING instructions cause scrolling, scrolling occurs only in the window.

The CLEAR, GCHAR, HCHAR, and VCHAR subprograms are not affected by the margin settings.

In Pattern and Text Modes, the margins can extend to the edges of the screen.

In the Split-Screen Modes, the margins can extend to the edges of the text portion of the screen.  The graphics portion of the screen is not affected by the margin settings.

High-Resolution and Multicolor Modes are not affected by the margin settings.

When you enter TI Extended BASIC II, the left and right margins are each set to 2, and the top and bottom margins are each set to 0.  When a program running in High-Resolution or Multicolor Mode ends, these default margin settings are restored.

When you change graphics mode, the default margin settings (2, 2, 0, 0) are restored.


EXAMPLES
100 CALL MARGINS(2,2,2,2)

Indents all four margins 2 spaces.


100 CALL MARGINS(5,5,8,8)

Indents the left and right margins 5 spaces; indents the top and bottom margins 8 spaces.

MAX function--Maximum

Format

MAX(numeric-expression1,numeric-expression2)

Type

Numeric (REAL or INTEGER)

Purpose

The MAX function returns the larger value of two numeric-expressions.

MAX is the opposite of the MIN function.

Cross Reference

MIN

------------------------------------------------------------------------

If the values of the numeric-expressions are equal, MAX returns that value.

EXAMPLES
| 100 PRINT MAX(3,8)

Prints 8.

| 100 F=MAX(3E12,1800000)

Sets F equal to 3E12.

| 100 G=MAX(-12,-4)

Sets G equal to -4.

| 100 A=7::B=-5
| 110 L=MAX(A,B)

Sets L equal to 7 when A=7 and B=-5.

MERGE

Format

MERGE ["]file-specification["]

Purpose

The MERGE command combines a program from an external storage device with the program currently in memory. MERGE is frequently used to combine several previously written program segments into one program.

Cross Reference

SAVE

----------------------------------------------------------------------

!o!    file-specification--The file-specification is a string constant
       that indicates the name of the program on the external device
       (see ~~Section X~~, "File Specifications"). The file-specification
       can optionally be enclosed in quotation marks.

The lines of the external program are inserted in line-number order among the
lines of the program in memory. If a line number in the external program
duplicates a line number in the program in memory, the new line replaces the
old line.

The MERGE command does not clear breakpoints.

A program on an external device can be merged only if it was saved with the
MERGE option of the SAVE command.

EXAMPLE
MERGE DSK1.SUB

Merges the program SUB into the program currently in memory.

## PROGRAM

Listed below is an example of how to merge programs. If the following program is saved on DSK1 as BOUNCE with the merge option, it can be merged with other programs.

```
|100 CALL CLEAR
|110 RANDOMIZE
|140 DEF RND50=INT(RND*50-25
)
|150 GOSUB 10000
|10000 FOR AA=1 TO 100
|10010 QQ=RND50
|10020 LL=RND50
|10030 CALL MOTION(#1,QQ,LL)
|10040 NEXT AA
|10050 RETURN
|SAVE "DSK1.BOUNCE",MERGE
|NEW
```

Place the following program into the computer's memory.

```
|120 CALL CHAR(96,"18183CFFF
F3C1818")
|130 CALL SPRITE(#1,96,7,92,
128)
|150 GOSUB 500
|160 STOP
```

Now merge BOUNCE with the above program.

```
|MERGE DSK1.BOUNCE
```

The program that results from merging BOUNCE with the above program is shown here.

```
|LIST
100 CALL CLEAR
110 RANDOMIZE
120 CALL CHAR(96,"18183CFFFF
3C1818")
130 CALL SPRITE(#1,96,7,92,1
28)
140 DEF RND50=INT(RND*50-25)
150 GOSUB 10000
160 STOP
10000 FOR AA=1 TO 100
10010 QQ=RND50
10020 LL=RND50
10030 CALL MOTION(#1,QQ,LL)
10040 NEXT AA
10050 RETURN
```

Note that line 150 is from the program that was merged (BOUNCE), not from the program that was in memory.

MIN function--Minimum

Format

MIN(numeric-expression1,numeric-expression2)

Type

Numeric (REAL or INTEGER)

Purpose

The MIN function returns the smaller value of two numeric-expressions.
MIN is the opposite of the MAX function.

Cross Reference

MAX

----------------------------------------------------------------------------
If the values of the numeric-expressions are equal, MIN returns that value.

EXAMPLES
|100 PRINT MIN(3,8)

Prints 3.

|100 F=MIN(3E12,1800000)

Sets F equal to 1800000.

|100 G=MIN(-12,-4)

Sets G equal to -12.

|100 A=7::B=-5
|110 L=MIN(A,B)

Sets L equal to -5 when A=7 and B=-5.

MOTION subprogram

Format

CALL MOTION(#sprite-number,vertical-velocity,horizontal-velocity[,...])

Purpose

The MOTION subprogram enables you to change the velocity of one or more sprites.

Cross Reference

SPRITE

---

!o! sprite-number--The sprite-number is a numeric expression whose value specifies the number of a sprite as assigned by the SPRITE subprogram.

!o! vertical-velocity, horizontal-velocity--The vertical- and horizontal-velocity are numeric expressions whose values range from -128 to 127. If both values are zero, the sprite is stationary. The speed of a sprite is in direct linear proportion to the absolute value of the specified velocity.

A positive vertical-velocity causes the sprite to move toward the top of the screen; a negative vertical-velocity causes the sprite to move toward the bottom of the screen.

A positive horizontal-velocity causes the sprite to move to the right; a negative horizontal-velocity causes the sprite to move to the left.

If neither the vertical- nor horizontal-velocity are zero, the sprite moves at an angle in a direction and at a speed determined by the velocity values.

When a moving sprite reaches an edge of the screen, it disappears. The sprite reappears in the corresponding position at the opposite edge of the screen.

PROGRAM

The following program illustrates a use of the MOTION subprogram.

```
|100 CALL CLEAR
|110 CALL.SPRITE(#1,33,5,92,
124)
|120 FOR XVEL=-16 TO 16 STEP
 2
|130 FOR YVEL=-16 TO 16 STEP
 2
|140 DISPLAY AT(12,11):XVEL;
YVEL
|150 CALL MOTION(#1,YVEL,XVE
L)
|160 NEXT YVEL
|170 NEXT XVEL
```

Line 110 creates a sprite.

Lines 120 and 130 set values for the motion of the sprite.

Line 140 displays the current values of the motion of the sprite.

Line 150 sets the sprite in motion.

Lines 160 and 170 complete the loops that set the values for the motion of the sprite.

NEW

Format

NEW

Purpose

The NEW command erases the program currently in memory, so that you can enter a new program.

----------------------------------------------------------------------

The NEW command restores the computer to the condition it was in when you selected TI Extended BASIC II from the main selection list, with the following exceptions:

!o!     Memory allocated by the INIT subprogram is not returned to the memory area available to Extended BASIC II.

!o!     Assembly-language subprograms loaded by the LOAD subprogram remain in memory.

NEW restores all other default values, closes any open files, erases all variable values and names, and cancels any BREAK or TRACE commands in effect.

NEXT

## Format

NEXT <u>control-variable</u>

## Purpose

The NEXT instruction marks the end of a FOR-NEXT loop.

You can use NEXT as either a program statement or a command.

## Cross Reference

FOR TO

---

!o!    <u>control-variable</u>--The <u>control-variable</u> is the same
       <u>control-variable</u> that appears in the corresponding FOR TO
       instruction.

The NEXT instruction is always paired with a FOR TO instruction to form a
FOR-NEXT loop (see FOR TO).

A NEXT statement cannot be part of an IF THEN statement.

If NEXT is used as a command, it must be part of a multiple-statement line. A
FOR TO instruction must precede it on the same line.

## PROGRAM
The following program illustrates a use of the NEXT statement in lines 130 and
140.

```
|100 TOTAL=0
|110 FOR COUNT=10 TO 0 STEP
-2
|120 TOTAL=TOTAL+COUNT
|130 NEXT COUNT
|140 FOR DELAY=1 TO 100::NEX
T DELAY
|150 PRINT TOTAL,COUNT;DELAY
|RUN
 30            -2  101
```

NUMBER

Format

;§NUMBER§; [initial-line-number][,increment]
;§NUM  §;

Purpose

The NUMBER command puts the computer in Number Mode, so that it automatically generates line numbers for your program.

--------------------------------------------------------------------------

!o!     initial-line-number--If you enter an initial-line-number, the first line number displayed is one you specify. If you do not specify an initial-line-number, the computer starts with line number 100.

!o!     increment--Succeeding line numbers are generated by adding the value of the numeric expression increment to the previous line number. Note that to specify an increment only (without specifying an initial-line-number), you must precede the increment with a comma. The default increment is 10.

If a line number generated by the NUMBER command is the number of a line already in the program in memory, the existing program line is displayed with the line number. To indicate that the displayed line is an existing program line, the prompt symbol (|) that normally appears to the left of the line number is not displayed. When the computer displays an existing program line, you can either edit the line or press ENTER to leave the line unchanged.

If you enter a program line that contains an error, the appropriate error message is displayed, and the same line number appears again, enabling you to retype the line correctly.

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

To leave Number Mode, press CLEAR (FCTN 4). If the computer is displaying only a line number (that is, a line number not followed by any characters), you can leave Number Mode by pressing ENTER, UP ARROW, DOWN ARROW, PROC'D, BEGIN, AID, or BACK.

Special Editing Keys in Number Mode

In Number Mode, you can use the editing keys whether you are changing existing program lines or entering new ones.

LEFT ARROW (FCTN S)--Pressing LEFT ARROW moves the cursor one character position to the left. When the cursor moves over a character, it does not change or delete it.

RIGHT ARROW (FCTN D)--Pressing RIGHT ARROW moves the cursor one character position to the right. When the cursor moves over a character, it does not change or delete it.

INS (FCTN 2)--Pressing INS enables you to insert characters at the cursor position. Characters that you type are inserted until you press one of the other special editing keys. The character at the cursor position and all characters to the right of the cursor move to the right as you type. You may lose characters if they move so far to the right that they are no longer in the program line.

DEL (FCTN 1)--Pressing DEL deletes the character in the cursor position. All characters to the right of the cursor move to the left.

ERASE (FCTN 3)--Pressing ERASE erases the program line currently displayed (including the line number). The program line is erased only from the screen, not from memory.

REDO (FCTN 8)--Pressing REDO causes the program line or other text most recently input to be displayed. This can be especially helpful if you make an error while editing a program line, causing the computer not to accept it. Pressing REDO displays the original line so that you can make corrections without having to retype the entire line. When you press REDO, the computer leaves Number Mode and enters Edit Mode.

CLEAR (FCTN 4)--Pressing CLEAR causes the computer to leave Number Mode. If you were entering a new program line, it is not accepted. If you were changing an existing program line, any changes that you made are ignored.

ENTER--If you press ENTER when the computer is displaying only a line number (that is, a line number not followed by any characters), the computer leaves Number Mode. If the line number is the number of an existing program line, that program line is not changed or deleted.

If you press ENTER when the computer is displaying a line number followed by a program line, that line is accepted and the next line number is generated. The displayed line may be a new line that you have entered, an existing program line that you have not changed, or an existing program line that you have edited.

UP ARROW (FCTN E)--UP ARROW works exactly the same as ENTER in Number Mode.

DOWN ARROW (FCTN X)--DOWN ARROW works exactly the same as ENTER in Number Mode.

EXAMPLE

In the following, what you type is UNDERLINED.  Press ENTER after each line.
NUM instructs the computer to number starting at 100 with increments of 10.

```
|NUM
|100 X=4
|110 Z=10
|120
|NUM 110
 110 Z=11
|120 PRINT (Y+X)/Z
|130
|NUM 105,5
|105 Y=7
 110 Z=11
|115
|LIST
100 X=4
105 Y=7
110 Z=11
120 PRINT (Y+X)/Z
```

NUM 110 instructs the computer to number starting at 110 with increments of
10.   Change line 110 to Z=11.

NUM 105,5 instructs the computer to number starting at line 105 with
increments of 5.   Line 110 already exists.

OLD

Format

OLD ["]file-specification["]

Purpose

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

Cross Reference

SAVE

-----------------------------------------------------------------------

!o!    file-specification—The file-specification indicates the name of
       the program to be loaded from the external device (see Section
       X, "File Specifications"). The file-specification, a string
       constant, can optionally be enclosed in quotation marks.

The program to be loaded can be one of the following:

   !o!    A saved TI Extended BASIC II program.

   !o!    A file in DISPLAY VARIABLE 80 format, created by the LIST
          command or a text editing or word processing program.

   !o!    A specially prepared assembly-language program that executes
          automatically when it is loaded.

Before the program is loaded, all open files are closed. The program
currently in memory is erased after the program begins to load.

For more information see Section X, "Loading an Existing Program." *or page X*

Protected and Unprotected Programs

To execute an unprotected Extended BASIC II program that has been loaded into
memory, enter the RUN command when the cursor appears. You can use the LIST
command to display the program or any portion of the program.

If the program was saved using the PROTECTED option of the SAVE command, it
starts executing by itself when it is loaded. When the program ends (either
normally or because of an error) or stops at a breakpoint, it is erased from
memory.

## EXAMPLES
|OLD CS1

Displays instructions and then loads into the computer's memory a program from a cassette recorder.

|OLD "DSK1.MYPROG"

Loads into the computer's memory the program MYPROG from the diskette in disk drive one.

|OLD DSK.DISK3.UPDATE83

Loads into the computer's memory the program UPDATE83 from the diskette named DISK3.

ON BREAK

Format

ON BREAK STOP
ON BREAK NEXT

Purpose

The ON BREAK statement enables you to specify the action you want the computer
to take when either a breakpoint is encountered or CLEAR (FCTN 4) is pressed.

Cross Reference

BREAK

---

!o!   STOP--If you enter the STOP option, or if your program does not
      include an ON BREAK statement, program execution stops when a
      breakpoint is encountered or CLEAR is pressed.

!o!   NEXT--If you enter the NEXT option, program execution continues
      normally (with the next program statement) when a breakpoint is
      encountered or CLEAR is pressed.  If you press CLEAR while the
      computer is performing an input or output operation with certain
      external devices, an error condition occurs, causing the program
      to halt.  When the NEXT option is in effect, pressing QUIT (FCTN
      =) is the only way to interrupt your program.  However, pressing
      QUIT erases the program in memory and causes you to exit from TI
      Extended BASIC II without closing any open files, possibly
      causing the loss of data in those files.

ON BREAK does not affect a breakpoint that occurs when a BREAK statement with
no line-number-list is encountered in a program.

PROGRAM
The following program illustrates the use of ON BREAK.

```
|100 CALL CLEAR
|110 BREAK 150
|120 ON BREAK NEXT
|130 BREAK
|140 FOR A=1 TO 50
|150 PRINT "CLEAR IS DISABLE
D."
|160 NEXT A
|170 ON BREAK STOP
|180 FOR A=1 TO 50
|190 PRINT "NOW IT WORKS."
|200 NEXT ·A
```

Line 110 sets a breakpoint at line 150.

Line 120 sets breakpoint handling to go to the next line.

A breakpoint occurs at line 130 despite line 120, because no line number has been specified after BREAK.  Enter CONTINUE.

No breakpoint occurs at line 150 because of line 120; CLEAR has no effect during the execution of lines 140 through 160 because of line 120.  Line 170 restores the normal use of CLEAR.

ON ERROR

Format

;§ON ERROR STOP§;
;§ON ERROR line-number§;

Purpose

The ON ERROR statement enables you to specify the action you want the computer to take if a program error occurs.

Cross Reference

ERR, GOSUB, RETURN

------------------------------------------------------------------------

!o!    STOP—If you enter the STOP option, or if your program does not
       include an ON ERROR statement, program execution stops when a
       program error occurs.

!o!    line-number—If you enter a line-number, a program error causes
       program control to be transferred to the subroutine that begins
       at the specified line-number.   A RETURN statement in the
       subroutine returns control to a specified program statement.

When an error transfers control to a subroutine, the line-number option is
cancelled.  If you wish to restore it, your program must execute an ON ERROR
line-number statement again.

The ON ERROR line-number statement does not transfer control when the error is
caused by a RUN statement.

PROGRAM
The following program illustrates the use of ON ERROR.

```
|100 CALL CLEAR
|110 DATA "A","4","B","C"
|120 ON ERROR 190
|130 FOR G=1 TO 4
|140 READ X$
|150 X=VAL(X$)
|160 PRINT X;"SQUARED IS";X*
·X
|170 NEXT G
|180 STOP
|190 REM ERROR SUBROUTINE
|200 ON ERROR 230
|210 X$="5"
|220 RETURN
|230 REM SECOND ERROR
|240 CALL ERR(CODE,TYPE,SEVE
R,LINE)
|250 PRINT "ERROR";CODE;" IN
 LINE";LINE
|260 RETURN 170
```

Line 120 causes any error to pass control to line 190.

Line 130 begins a loop.  An error occurs in line 150 and control passes to line 190.

Line 200 causes the next error to pass control to line 230.

Line 210 changes the value of X$ to an acceptable value.  Line 220 returns control to the line in which the error occurred (line 150).

The second time an error occurs, the SECOND ERROR subroutine is called because of line 200.  Line 240 obtains specific information about the error by using CALL ERR.  Line 250 reports the nature of the error, and line 260 returns control to line 170 of the main program, which begins the next iteration of the loop.

When the third error occurs, the message Bad argument in 150 is displayed because the program does not specify what action to take if another error occurs.  Program execution ceases.

ON GOSUB

Format

ON numeric-expression ;§GOSUB §; line-number-list
                      ;§GO SUB§;

Purpose

The ON GOSUB statement enables you to transfer conditional program control to one of several subroutines.

Cross Reference

GOSUB, RETURN

------------------------------------------------------------------------

!o!   numeric-expression--The value of the numeric-expression determines to which of the line numbers in the line-number-list program control is transferred.

If the value of the numeric-expression is 1, program control is transferred to the subroutine that begins at the program statement specified by the first line number in the line-number-list; if the value of the numeric-expression is 2, program control is transferred to the subroutine that begins at the program statement specified by the second line number in the line-number-list; and so on.

If necessary, the value of the numeric-expression is rounded to the nearest integer. The value of the numeric-expression must be greater than or equal to 1 and less than or equal to the number of line numbers in the line-number-list.

!o!   line-number-list--The line-number-list consists of one or more line numbers separated by commas. Each line number specifies a program statement at which a subroutine begins.

Use a RETURN statement to return program control to the statement immediately following the ON GOSUB statement that called the subroutine.

To avoid unexpected results, it is recommended that you exercise special care if you use ON GOSUB to transfer control to or from a subprogram or into a FOR-NEXT loop.

EXAMPLES
```
|100 ON X GOSUB 1000,2000,30
0
```

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
|100 ON P-4 GOSUB 200,250,30
0,800,170
```

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.

PROGRAM
The following program illustrates a use of ON GOSUB.

```
|100 CALL CLEAR
|110 DISPLAY AT(11,1):"CHOOS
E ONE OF THE FOLLOWING:"
|120 DISPLAY AT(13,1):"1  AD
D TWO NUMBERS."
|130 DISPLAY AT(14,1):"2  MU
LTIPLY TWO NUMBERS."
|140 DISPLAY AT(15,1):"3  SU
BTRACT TWO NUMBERS."
|150 DISPLAY AT(16,1):"4  EX
IT PROGRAM."
|160 DISPLAY AT(20,1):"YOUR
CHOICE:"
|170 DISPLAY AT(22,2):"FIRST
 NUMBER:"
|180 DISPLAY AT(23,1):"SECON
D NUMBER:"
|190 CALL MARGINS(2,2,0,0)
|200 ACCEPT AT(20,14)VALIDAT
E(DIGIT):CHOICE
|210 IF CHOICE 1 OR CHOICE|4
 THEN 200
|220 IF CHOICE=4 THEN STOP
|230 ACCEPT AT(22,16)VALIDAT
E(NUMERIC):FIRST
|240 ACCEPT AT(23,16)VALIDAT
E(NUMERIC):SECOND
|250 CALL MARGINS(2,2,0,16)
|260 ON CHOICE GOSUB 280,300
,320
|270 GOTO 190
```

```
|280 DISPLAY AT(3,1)ERASE AL
L:FIRST;"PLUS";SECOND;"EQUAL
S";FIRST+SECOND
|290 RETURN
|300 DISPLAY AT(3,1)ERASE AL
L:FIRST;"TIMES";SECOND;"EQUA
LS";FIRST*SECOND
|310 RETURN
|320 DISPLAY AT(3,1)ERASE AL
L:FIRST;"MINUS";SECOND;"EQUA
LS";FIRST-SECOND
|330 RETURN
```

Line 260 determines where to go according to the value of CHOICE.

ON GOTO

Format

ON numeric-expression ;§GOTO §; line-number-list
                ;§GO TO§;

Purpose

The ON GOTO statement enables you to transfer unconditional program control to one of several program statements.

Cross Reference

GOTO

--------------------------------------------------------------------------

!o!    numeric-expression--The value of the numeric-expression determines to which of the line numbers in the line-number-list program control is transferred. If the value of the numeric-expression is 1, program control is transferred to the program statement specified by the first line number in the line-number-list; if the value of the numeric-expression is 2, program control is transferred to the program statement specified by the second line number in the line-number-list; and so on.

        If necessary, the value of the numeric-expression is rounded to the nearest integer. The value of the numeric-expression must be greater than or equal to 1 and less than or equal to the number of line numbers in the line-number-list.

!o!    line-number-list--The line-number-list consists of one or more line numbers separated by commas. Each line number specifies a program statement.

To avoid unexpected results, it is recommended that you exercise care if you use ON GOTO to transfer control to or from a subroutine or a subprogram or into a FOR-NEXT loop.

## EXAMPLES

```
|100 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.  The equivalent statement using an IF-THEN-ELSE statement is IF X=1 THEN 1000 ELSE IF X=2 THEN 2000 ELSE IF X=3 THEN 300 ELSE PRINT "ERROR!"::STOP.

```
|100 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.

## PROGRAM

The following program illustrates a use of ON __ GOTO.  Line 260 determines where to go according to the value of CHOICE.

```
|100 CALL CLEAR
|110 DISPLAY AT(11,1):"CHOOS
E ONE OF THE FOLLOWING:"
|120 DISPLAY AT(13,1):"1  AD
D TWO NUMBERS."
|130 DISPLAY AT(14,1):"2  MU
LTIPLY TWO NUMBERS."
|140 DISPLAY AT(15,1):"3  SU
BTRACT TWO NUMBERS."
|150 DISPLAY AT(16,1):"4  EX
IT PROGRAM."
|160 DISPLAY AT(20,1):"YOUR
CHOICE:"
|170 DISPLAY AT(22,2):"FIRST
 NUMBER:"
|180 DISPLAY AT(23,1):"SECON
D NUMBER:"
|190 CALL MARGINS(2,2,0,0)
|200 ACCEPT AT(20,14)VALIDAT
E(DIGIT):CHOICE
|210 IF CHOICE 1 OR CHOICE|4
 THEN 200
|220 IF CHOICE=4 THEN STOP
|230 ACCEPT AT(22,16)VALIDAT
E(NUMERIC):FIRST
|240 ACCEPT AT(23,16)VALIDAT
E(NUMERIC):SECOND
|250 CALL MARGINS(2,2,0,16)
|260 ON CHOICE GOTO 270,290,
310
```

```
|270 DISPLAY AT(3,1)ERASE AL
L:FIRST;"PLUS";SECOND;"EQUAL
S";FIRST+SECOND
|280 GOTO 190
|290 DISPLAY AT(3,1)ERASE AL
L:FIRST;"TIMES";SECOND;"EQUA
LS";FIRST*SECOND
|300 GOTO 190
|310 DISPLAY AT(3,1)ERASE AL
L:FIRST;"MINUS";SECOND;"EQUA
LS";FIRST-SECOND
|320 GOTO 190
```

ON WARNING

Format

```
ON WARNING ;§PRINT§;
           ;§STOP §;
           ;§NEXT §;
```

Purpose

The ON WARNING statement enables you to specify the action you want the computer to take if a warning condition occurs during the execution of your program.

------------------------------------------------------------------------

A warning, a condition caused by invalid input or output, does not normally cause program execution to be terminated.

> !o!     PRINT--If you enter the PRINT option, or if your program does not include an ON WARNING statement, the computer displays a warning message when a warning condition occurs during program execution.

> !o!     STOP--If you enter the STOP option, program execution stops when a warning condition occurs during program execution.

> !o!     NEXT--If you enter the NEXT option, program execution continues normally when a warning condition occurs and no warning message is displayed.  Normally, execution continues beginning with the next program statement; however, if the cause of the warning is an invalid response to an INPUT statement, program execution continues beginning with that same INPUT statement.

You may have multiple ON WARNING statements in the same program.

PROGRAM
The following program illustrates the use of ON WARNING.

```
|100 CALL CLEAR
|110 ON WARNING NEXT
|120 PRINT 120,5/0
|130 ON WARNING PRINT
|140 PRINT 140,5/0
|150 ON WARNING STOP
|160 PRINT 160,5/0
|170 PRINT 170
|RUN
 120           9.99999E+**
 140
* WARNING
  NUMERIC OVERFLOW IN 140
                9.99999E+**
 160
* WARNING
  NUMERIC OVERFLOW IN 160
```

Line 110 sets warning handling to go to the next line.  Line 120 therefore prints the result without any message.

Line 130 sets warning handling to the default, printing the message and then continuing execution.  Line 140 therefore prints 140, then the warning, and then continues.

Line 150 sets warning handling to print the warning message and then stop execution.  Line 160 therefore prints 160 and the warning message and then stops.

!o!     file-type--The file-type specifies the format of data in the
        file.

> INTERNAL--The computer transfers data in binary format.
> This is the most efficient method of sending data.
>
> DISPLAY--The computer transfers data in ASCII format.
> DISPLAY files can only use FIXED records of 64 and 128.  If
> no file-type is specified in OPEN, the default is DISPLAY.
>
> DISPLAY type files require a special kind of output
> record.  Each element in the PRINT field must be separated
> by a comma enclosed in quotation marks.  The comma serves
> as a field separator in the file.  The omission of this
> comma causes an I/O error.  Note: This is not the same as a
> print separator, which must be inserted between an element
> in the PRINT field and the field separator.

!o!     open-mode--The open-mode specifies the input/output operations
        that can be performed on the file.

> INPUT--The computer can only read data from the file.
>
> OUTPUT--The computer can only write data to the file.
>
> UPDATE--The computer can both read from and write to the
> file.
>
> APPEND--The computer can only write data and only at the
> end of the file; records already in the file cannot be
> accessed.

If you open an existing file for OUTPUT, the data items you
write to the file replaces those currently in the file.

If you do not specify an open-mode, it is assumed to be UPDATE.

!o!     record-type--The record-type specifies whether the records in
        the file are FIXED (all of the same length) or VARIABLE (of
        varying lengths).

SEQUENTIAL files can have FIXED or VARIABLE records.  If you do
not specify the record-type of a SEQUENTIAL file, it is assumed
to be VARIABLE.

RELATIVE files must have FIXED records.  If you do not specify
the record-type of a RELATIVE file, it is assumed to be FIXED.

!o!     record-length--You can optionally specify the length of records
        in the file.  Record-length is a numeric expression, the value
        of which specifies the fixed size (for FIXED records) or maximum
        size (for VARIABLE records) of each record.

If you do not specify a record-length, its value is supplied by
the peripheral.

OPEN

## Format

OPEN #file-number:file-specification[,file-organization[ size]]
[,file-type][,open-mode][,record-type[ record-length]]

## Purpose

The OPEN instruction establishes an association between the computer and an
external device, enabling you to store, retrieve, and process data.

## Cross Reference

CLOSE, INPUT, PRINT

------------------------------------------------------------------

!o!  file-number--The file-number is a numeric expression having a
     value between 1 and 255.  The file-number is assigned to the
     external file or device indicated by the file-specification, so
     that input/output processing instructions may refer to the file
     by its file-number.  While a file is open, its file-number
     cannot be assigned to another file.  However, you may have more
     than one file open to a device at one time.  File-number 0
     always refers to the keyboard and screen of your computer, and
     is always open.  You cannot open or close file-number 0.

     If necessary, the file-number is rounded to the nearest integer.

!o!  file-specification--The file-specification is a string
     expression; if you use a string constant, you must enclose it in
     quotation marks.  For more information, see "File
     Specifications," page XX.

## Options

The following options may be entered in any order.

!o!  file-organization--The file-organization specifies whether
     records are to be accessed sequentially or randomly.  Enter
     SEQUENTIAL for sequential access, or RELATIVE for random
     access.  Records in a sequential-access file are read or written
     in sequence from beginning to end.  Records in a random-access
     (relative-record) file can be accessed in any order (they can be
     processed randomly or sequentially).

     If you do not specify a file-organization, it is assumed to be
     SEQUENTIAL.

!o!  size--You can optionally specify the initial size of the file.
     Size is a numeric expression, the value of which specifies the
     initial number of records in the file.  Note: The size option
     cannot be used with all peripherals.

If you open a file that does not exist, a file is created with the options you specify.  If you open a file that does exist, the options you specify must be the same as the options that you specified when you created the file, except that a file with FIXED records can be opened as either SEQUENTIAL or RELATIVE, regardless of the file-organization that you specified when you created the file.

For more information about the options available with a particular device, refer to the owner's manual that comes with that device.

EXAMPLES
```
|100 OPEN #1:"CS1",OUTPUT,FI
XED
```

Opens a file on cassette.  The file is SEQUENTIAL, with data stored in DISPLAY format.  The file is opened in OUTPUT mode with FIXED length records that have a length of 64 bytes.

```
|300 OPEN #23:"DSK.MYDISK.X"
,RELATIVE 100,INTERNAL,FIXED
,UPDATE
```

Opens a file named "X".  The file is on the diskette named MYDISK in whichever drive that diskette is located.  The file is RELATIVE, with data kept in INTERNAL format with FIXED length records that have a maximum length of 80 bytes.  The file is opened in UPDATE mode and starts with 100 records made available for it.

```
|100 OPEN #243:A$,INTERNAL
```

Where A$ equals "DSK2.ABC", assumes a file on the diskette in drive 2 with a name of ABC.  The file is SEQUENTIAL, with data kept in INTERNAL format. The file is opened in UPDATE mode with VARIABLE length records that have a maximum length of 80 bytes.

## PROGRAM

The following program emulates the use of the SIZE option in an OPEN statement.

```
|100 OPEN #1:"DSK1.LARGE",RE
LATIVE
|110 PRINT #1,REC 100:0
|120 CLOSE #1
|130 OPEN #1:"DSK1.LARGE",SE
QUENTIAL,FIXED
  •
  •
  •
|200 CLOSE #1
```

Line 100 opens a RELATIVE file on diskette.

Line 110 writes to the 100th record, thereby reserving space for 100 contiguous records.

Line 120 closes the file.

Line 130 reopens the file, this time with SEQUENTIAL file organization.

Line 200 closes the file.

OPTION BASE

## Format

```
OPTION BASE ;§0§;
            ;§1§;
```

## Purpose

The OPTION BASE statement enables you to set the lower limit of array subscripts.

## Cross Reference

DIM, INTEGER, REAL

------------------------------------------------------------------

You can use the OPTION BASE statement to specify a lower array-subscript limit of either 0 or 1.  If your program does not include an OPTION BASE statement, the lower limit is set to 0.

The OPTION BASE statement applies to every array in your program.  You can have only one OPTION BASE statement in a program.

If you do not set the lower array-subscript limit to 1, the computer reserves memory for element 0 of each dimension of each array.  To avoid reserving unnecessary memory, it is recommended that you set the lower limit to 1 if your program does not use element 0.

The OPTION BASE statement must have a lower line number than any DIM statement or any reference to an array in your program.  The OPTION BASE statement is evaluated during pre-scan and is not executed.

The OPTION BASE statement cannot be part of an IF THEN statement.

## EXAMPLE
100 OPTION BASE 1

Sets the lowest allowable subscript of all arrays to one.

PATTERN subprogram

Format

CALL PATTERN(#sprite-number,character-code[...])

Purpose

The PATTERN subprogram enables you to change the pattern of one or more sprites.

Cross Reference

CHAR, MAGNIFY, SPRITE

------------------------------------------------------------------------

!o!  sprite-number--The sprite-number is a numeric expression whose value specifies the number of the sprite as assigned in the SPRITE subprogram.

!o!  character-code--Character-code is a numeric expression with a value from 0-255, specifying the character number of the character you want to use as the pattern for a sprite.

If you use the MAGNIFY subprogram to change to double-sized sprites, the sprite definition includes the character specified by the character-code and three additional characters (see MAGNIFY).

PROGRAM

The following program illustrates the use of the PATTERN subprogram.

```
100 CALL CLEAR
110 CALL COLOR(12,16,16)
120 FOR A=19 TO 24
130 CALL HCHAR(A,1,120,32)
140 NEXT A
150 A$="01071821214141FFFF4
141212119070080E09884848282F
FFF8282848498E000"
160 B$="01061820305C4681814
246242C18070080601834246242B
181623A0C0418E000"
170 C$="0106182C24464281814
65C302018070080601804OC3A628
1814262243418E000"
180 CALL CHAR(244,A$,248,B$
,252,C$)
190 CALL SPRITE(#1,244,5,13
0,1,0,8)
200 CALL MAGNIFY(3)
210 FOR A=244 TO 252 STEP 4
220 CALL PATTERN(#1,A)
230 FOR DELAY=1 TO 15:: NEX
T DELAY
240 NEXT A
250 GOTO 210
        (Press CLEAR to stop the program.)
```

Lines 110 through 140 build a floor.

Lines 150 though 180 define characters 244 through 255.

Line 190 creates a sprite in the shape of a wheel and starts it moving to the right.

Line 200 makes the sprite double-sized.

Lines 210 through 250 make the spokes of the wheel appear to move as the character displayed is changed.

See also the third example of the SPRITE subprogram, on page X.

PEEK subprogram—Peek at CPU RAM

**Format**

```
CALL PEEK(address,numeric-variable-list
     [,"",address,numeric-variable-list[,...]])
```

**Purpose**

The PEEK subprogram enables you to ascertain the contents of specified CPU memory addresses.

You can use the PEEKV subprogram to ascertain the contents of VDP memory.

**Cross Reference**

LOAD, PEEKV, POKEV, VALHEX

--------------------------------------------------------------------------------

!o!    address—The address is a numeric expression whose value specifies the first CPU (Central Processing Unit) memory address at which you want to peek.

The address must have a value from -32768 to 32767 inclusive.

You can specify an address from 0 to 32767 inclusive by specifying the actual address.

You can specify an address from 32768 to 65535 inclusive by subtracting 65536 from the actual address.  This will result in a value from -32768 to -1 inclusive.

If you know the hexadecimal value of the address, you can use the VALHEX function to convert it to a decimal numeric expression, eliminating the need for manual calculations.

If necessary, the address is rounded to the nearest integer.

!o!    numeric-variable-list—The numeric-variable-list consists of one or more numeric-variables separated by commas.  Bytes of data starting from the specified CPU memory address are assigned sequentially to the numeric-variables in the numeric-variable-list.

One byte, with a value from 0 to 255 inclusive, is returned to each specified numeric-variable.

You can specify multiple <u>addresses</u> and <u>numeric-variable-lists</u> by entering a null string (two adjacent quotation marks) as a separator between a <u>numeric-variable-list</u> and the next <u>address</u>.

If you call the PEEK subprogram with invalid parameters, the computer may function erratically or cease to function entirely.  If this occurs, turn off the computer, wait several seconds, and then turn the computer back on again.

EXAMPLES
```
|100 CALL PEEK(8192,X1,X2,X3
,X4)
```

Returns the values in memory locations 8192, 8193, 8194, and 8195 in the variables X1, X2, X3, and X4, respectively.

```
|100 CALL PEEK(~~-4276~~,X,Y,Z,
",62435,A,B,C)
```

Returns the values in locations *61260, 61261,* and *61262* in X, Y, and Z, respectively; and the values in locations 62433, 62434, and 62435 in A, B, and C, respectively.

```
|100 CALL PEEK(VALHEX("4F55"
),V1,V2,V3)
```

Uses VALHEX to ascertain the decimal equivalent of the hexadecimal number 4F55, which is 20309.  Then the values in locations 20309, 20310, and 20311 are returned in V1, V2, and V3, respectively.

PROGRAM
The following program returns in A the number of the highest numbered sprite (#15) currently in use.  A zero is returned to B, because no sprites are defined after the DELSPRITE statement.

```
|100 CALL CLEAR
|110 CALL SPRITE(#15,33,7,10
0,100,0,0)
|120 CALL PEEK(VALHEX("837A"
),A)
|130 CALL DELSPRITE(ALL)
|140 CALL PEEK(VALHEX("837A"
),B)
|150 PRINT A,B
```

PEEKV subprogram--Peek at VDP RAM

Format

CALL PEEKV(address,numeric-variable-list
    [,"",address,numeric-variable-list[,...]])

Purpose

The PEEKV subprogram enables you to ascertain the contents of specified VDP memory addresses.

You can use the PEEK subprogram to ascertain the contents of CPU memory.

Cross Reference

LOAD, PEEK, POKEV, VALHEX

-----------------------------------------------------------------

!o!   address--The address is a numeric expression whose value specifies the first VDP (Video Display Processor) memory address at which you want to peek.

The address must have a value from 0 to 16383 inclusive.

If you know the hexadecimal value of the address (0000-3FFF), you can use the VALHEX function to convert it to a decimal numeric expression.

If necessary, the address is rounded to the nearest integer.

!o!   numeric-variable-list--The numeric-variable-list consists of one or more numeric-variables separated by commas.  Bytes of data starting from the specified VDP memory address are assigned sequentially to the numeric-variables in the numeric-variable-list.

One byte, with a value from 0 to 255 inclusive, is returned to each specified numeric-variable.

You can specify multiple addresses and numeric-variable-lists by entering a null string (two adjacent quotation marks) as a separator between a numeric-variable-list and the next address.

If you call the PEEKV subprogram with invalid parameters, the computer may function erratically.  If this occurs, turn off the computer, wait several seconds, then turn the computer back on.

2173L--Computer 99/8 BASIC Reference Guide--Final Draft  9/15/83

EXAMPLE
```
100 CALL PEEKV(6300,A1,A2,A
3)
```

Returns the values in locations 6300, 6301, and 6302 in A1, A2, and A3, respectively.

PROGRAMS
The following program gives an example of the use of PEEKV.

```
100 CALL CLEAR
110 CALL POKEV(32*12+16,66)
120 CALL PEEKV(32*12+16,A)
130 PRINT A
```

Line 110 pokes a "B" into a location that causes it to appear in the middle of the screen. Line 120 peeks at that location, and assigns the value found there (66) to A. *the variable*

The next program starts a sprite moving diagonally across the screen. Line 120 assigns the values of the row and column coordinates of the sprite to Y and X, respectively.

```
100 CALL CLEAR
110 CALL SPRITE(#1,33,5,100
,100,25,25)
120 CALL PEEKV(VALHEX("300"
),Y,X)
130 DISPLAY AT(24,1):Y;X
140 GOTO 120
```
          (Press CLEAR to stop the program.)

Page 163

PI function--Pi

**Format**

PI

**Type**

REAL

**Purpose**

The PI function returns the value of pi.

-----------------------------------------------------------------------

The value of pi is 3.14159265359.

EXAMPLE
T100 VOLUME=4/3*PI*6^3

Sets VOLUME equal to four-thirds times pi times six cubed, which is the volume
of a sphere with a radius of six.

POKEV subprogram--Poke to VDP RAM

Format

CALL POKEV(address,byte-list[,"",address,byte-list[,...]])

Purpose

The POKEV subprogram enables you to assign values directly to specified VDP memory addresses.

You can use the LOAD subprogram to assign values to CPU memory.

Cross Reference

LOAD, PEEK, PEEKV, VALHEX

---

!o!　　address--The address is a numeric expression whose value specifies the first VDP (Video Display Processor) memory address where data is to be poked.  If the byte-list specifies more than one byte of data, the bytes are assigned to sequential memory addresses starting with the address you specify.

　　　　The address must have a value from 0 to 16383 inclusive.

　　　　If you know the hexadecimal value of the address (0000-3FFF), you can use the VALHEX function to convert it to a decimal numeric expression.

　　　　If necessary, the address is rounded to the nearest integer.

!o!　　byte-list--The byte-list consists of one or more bytes of data, separated by commas, that are to be poked into VDP memory starting with the specified address.

　　　　Each byte in the byte-list must be a numeric expression with a value from 0 to 32767.  If the value of a byte is greater than 255, it is repeatedly reduced by 256 until it is less than 256. If necessary, a byte is rounded to the nearest integer.

You can specify multiple addresses and byte-lists by entering a null string (two adjacent quotation marks) as a separator between a byte-list and the next address.

If you call the POKEV subprogram with invalid parameters the computer may function erratically.  If this occurs, turn off the computer, wait several seconds, then turn the computer back on.

## EXAMPLES

|100 CALL POKEV(3333,233)

Pokes the value 233 into location 3333.


|100 CALL POKEV(13784,273)

Pokes the value 17 (273 reduced by 256 once) into location 13784.


|100 CALL POKEV(7343,246,"",
VALHEX("2E4F"),433)

Pokes the value 246 into location 7343, and uses VALHEX to ascertain the decimal equivalent of the hexadecimal number 2E4F (11855). The value 177 (433 reduced by 256 once) is then poked into this location.


## PROGRAM

The following program uses POKEV to display on the screen the characters that correspond to ASCII codes 65 through 208, at the location specified by the value of R*32+C.

```
|100 CALL CLEAR::X=65
|110 FOR R=0 TO 23
|120 FOR C=0 TO 31 STEP 6
|130 CALL POKEV(R*32+C,X)
|140 X=X+1
|150 NEXT C
|160 NEXT R
```

POS function--Position

Format

POS(string-expression,substring,numeric-expression)

Type

INTEGER

Purpose

The POS function returns the position of the first occurrence of a substring within a specified string.

------------------------------------------------------------------------

!o!  string-expression--The string-expression specifies the string within which you are seeking the substring.  If you use a string constant, it must be enclosed in quotation marks.

!o!  substring--The substring is the segment (of the string-expression) you are trying to locate.  The substring is a string expression; if you use a string constant, it must be enclosed in quotation marks.

!o!  numeric-expression--The value of the numeric-expression specifies the character position in the string-expression where the search for the substring begins.

If necessary, the value of the numeric-expression is rounded to the nearest integer.  The value of the numeric-expression must be greater than zero.

If the substring is present within the string-expression, POS returns the number of the character position (within the string-expression) of the first character of the substring.

If the substring is not present, or if the value of the numeric-expression is greater than the number of characters in the string-expression, POS returns a zero.

EXAMPLES
```
|100 X=POS("PAN","A",1)
```

Sets X equal to 2 because A is the second letter in PAN.

```
|100 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.          •

```
|100 Z=POS("PAN","A",3)
```

Sets Z equal to 0 because A was not in the part of PAN that was searched.

```
|100 R=POS("PABNAN","AN",1)
```

Sets R equal to 5 because the first occurrence of AN starts with the A in the fifth position in PABNAN.

PROGRAM
The following program illustrates a use of POS.  Input is searched for spaces, and is then printed with each word on a single line.

```
|100 CALL CLEAR
|110 PRINT "ENTER A SENTENCE
."
|120 LINPUT X$
|130 S=POS(X$," ",1)
|140 IF S=0 THEN PRINT X$::P
RINT::GOTO 110
|150 Y$=SEG$(X$,1,S)::PRINT
Y$
|160 X$=SEG$(X$,S+1,LEN(X$))
|170 GOTO 130
```
        (Press CLEAR to stop the program)

POSITION subprogram


**Format**

CALL POSITION(#sprite-number,numeric-variable1,numeric-variable2[,...])


**Purpose**

The POSITION subprogram enables you to ascertain the current position of one or more sprites.

**Cross Reference**

SPRITE

--------------------------------------------------------------------------

    !o!    sprite-number--The sprite-number is a numeric expression whose
           value specifies the number of the sprite as assigned in the
           SPRITE subprogram.

    !o!    numeric-variables--The current screen position of a sprite is
           returned as two numeric-variables representing the pixel-row and
           the pixel-column, respectively, specifying the position of a
           screen pixel.

The screen position of the pixel in the upper-left corner of a sprite is considered to be the position of that sprite.

Note that a sprite in motion continues to move during and following the execution of the POSITION subprogram. Remember to allow for this continued motion in your program.


EXAMPLE
1100 CALL POSITION(#1,Y,X)

Returns the position of the upper left corner of sprite #1.

Also see the third example of the SPRITE subprogram, on page X.

PRINT

Format

Print to the Screen
        PRINT [print-list]
Print to a File (or Device)
        PRINT #file-number[,REC record-number][:print-list]

Purpose

The PRINT instruction enables you to display data items on the screen or print
them to an external device.

You can use PRINT as either a program statement or a command.

Cross Reference

DISPLAY, OPEN, PRINT USING, TAB

---------------------------------------------------------------------------

        !o!    print-list--The print-list consists of one or more print-items
               (items to be printed or displayed) separated by
               print-separators.  A PRINT instruction without a print-list
               advances the print position to the first position of the next
               record.  This has the effect of printing a blank record, unless
               the preceding PRINT instruction ended with a print-separator.

               The numeric and/or string expressions in the print-list can be
               constants and/or variables.

Print-items are the numeric and string expressions to be printed.  Any
function is also a valid print-item.

Print-separators are the punctuation (commas, semicolons, and colons) between
print-items specifying the placement of the print-items in the print record.

Printing to the Screen

Each print-item is displayed in the bottom row of the screen window defined by
the margins, starting from the far left column of the window.  Before a new
line is displayed at the bottom of the window, the entire contents of the
window (excluding sprites) scroll up one line to make room for the new line.
The contents of the top line of the window scroll off the screen and are
discarded.

Each line on the screen is treated as one print record.  The record length of the screen is the width of the window.

In High-Resolution and Multicolor Modes, the print-items "displayed" on the screen are not visible.  PRINT cannot access the graphics portion of the screen in the Split-Screen Modes.

## Printing to a File

> !o!  file-number--If you include an optional file-number, the print-list is sent to the specified device.  The file-number is a numeric expression whose value specifies the number of the file as assigned in its OPEN instruction.  You cannot print to a file opened in INPUT mode.
>
> If you do not specify a file-number (or if you specify file-number 0), the print-list is displayed on the screen.
>
> !o!  record-number--If you use the REC option, the record-number is a numeric expression whose value specifies the number of the record in which you want to print the print-list.  The records in a file are numbered sequentially, starting with zero.  The REC option can be used only with a file opened for RELATIVE access.

If you print to a file opened in INTERNAL format with FIXED records, each record is filled with trailing binary zeros, if necessary, to bring it to its specified length.  If a record is longer than the record length of the file, it is truncated (extra characters are discarded).

For more information about printing to a particular device, refer to the owner's manual that comes with that device.

## Printing Numbers: INTERNAL Files

The amount of memory space allocated to a number printed to a file opened in INTERNAL format varies according to its data-type.  An INTEGER is always allocated 3 bytes, whereas a REAL number is always allocated 9 bytes.

Note that if you print an INTEGER value to a file, you cannot access that file on a Home Computer (such as the TI-99/4A) that does not support the INTEGER data-type.  You can circumvent this by converting all INTEGER variables and functions to REAL variables before printing them to a file.

## Printing Numbers: The Screen and DISPLAY Files

The format of a number printed to the screen or to a file opened in DISPLAY format varies according to the characteristics of the number.

Positive numbers and zero are printed with a leading space (instead of a plus sign); negative numbers are printed with a leading minus sign.  All numbers are printed with a trailing space.

Numbers are printed in either decimal form or scientific notation, according to these rules:

!o!     All numbers with 10 or fewer digits are printed in decimal form.

!o!     REAL numbers with more than 10 digits are printed in scientific notation only if they can be presented with more significant digits in scientific notation than in decimal form.  If printed in decimal form, all digits beyond the tenth are omitted.

If a number is printed in decimal form, the following rules apply:

!o!     INTEGER numbers and REAL numbers with no decimal portion are printed without decimal points.

!o!     REAL numbers are printed with decimal points in the proper position.  If the number has more than 10 digits, it is rounded to 10 digits.  A zero is not printed by itself to the left of the decimal point.  Trailing zeros after the decimal point are omitted.

If a number is printed in scientific notation, the following rules apply:

!o!     The format is <u>mantissaEexponent</u>.

!o!     The <u>mantissa</u> is printed with six or fewer digits, with one digit to the left of the decimal point.

!o!     Trailing zeros are omitted after the decimal point of the <u>mantissa</u>.

!o!     If there are more than five digits after the decimal point of the <u>mantissa</u>, the fifth digit is rounded.

!o!     The <u>exponent</u> is a two-digit number displayed with a plus or minus sign.

!o!     If you attempt to print a number with an <u>exponent</u> greater than 99 or less than -99, the computer prints two asterisks (**) following the sign of the exponent.

## Printing Strings

A string constant in a print-list must be enclosed in quotation marks. A quotation mark within a string constant is represented by two adjacent quotation marks.

A string printed to a file opened in INTERNAL format has a length one greater than the length of the string.

When a string is printed to the screen or to a file opened in DISPLAY format, no leading or trailing spaces are added to the string.

## Print/Separators

At least one print/separator must be placed between adjacent print/items in the print-list. Valid print/separators are the semicolon (;), the colon (:), and the comma (,).

> !o! A semicolon (;) print/separator causes the next print/item to print immediately after the current print/item.

> !o! A colon (:) print/separator causes the next print/item to print at the beginning of the next record. Each extra colon causes one blank record to be printed. Consecutive colons used as print separators must be divided by a space. Otherwise, they are treated as a statement separator symbol.

> !o! If you print to the screen or to a file opened in DISPLAY format, a comma (,) print/separator causes the next print/item to print at the beginning of the next "zone." Print records are divided into 14-character zones; the number of zones in a print record varies according to its record length.

If you print to a file opened in INTERNAL format, a comma print/separator has the same effect as a semicolon print/separator.

If a print/separator would have the effect of splitting the next print/item between two records, the print/item is moved to the beginning of the following record. However, if discarding the trailing space from a numeric print/item allows it to fit in the current record, the number is printed in the current record without its trailing space.

If the print/list ends with a print/separator, the computer is placed in a print-pending condition. Unless the next PRINT instruction includes the REC option, it is considered to be a continuation of the current PRINT instruction. RESTORE #file-number terminates a print-pending condition.

If the print/list is not terminated by a print/separator, the computer considers the current record complete when all the print/items in the print/list are printed. The first print/item in the next PRINT instruction begins in the next record.

EXAMPLES
|100 PRINT

Causes a blank line to appear on the display screen.


|100 PRINT "THE ANSWER IS";A
NSWER

Causes the string constant THE ANSWER IS to be printed on the display screen,
followed immediately by the value of ANSWER.  If ANSWER is positive, there
will be a blank for the positive sign after IS.


|100 PRINT X:Y/2

Causes the value of X to be printed on a line and the value of Y/2 to be
printed on the next line.


|100 PRINT #12,REC 7:A

Causes the value of A to be printed on the eighth record of the file that was
opened as number 12 with RELATIVE file organization.  (Record number 0 is the
first record.)


|100 PRINT #32:A,B,C,

Causes the values of A, B, and C to be printed on the next record of the file
that was opened as number 32.  The final comma creates a pending print-
condition.  The next PRINT statement directed to file number 32 will print on
the same record as this PRINT statement unless it specifies a record, or a
RESTORE #32 statement is executed, thereby closing the pending print-
condition.


|100 PRINT #1,REC 3:A,B
|150 PRINT #1:C,D

Causes A and B to be printed in record 3 of the file that was opened as number
1.  PRINT #1:C,D causes C and D to be printed in record 4 of the same file.

PROGRAM
The following program prints out values in various positions on the screen.

```
|100 CALL CLEAR
|110 PRINT 1;2;3;4;5;6;7;8;9
|120 PRINT 1,2,3,4,5,6
|130 PRINT 1:2:3
|140 PRINT
|150 PRINT 1;2;3;
|160 PRINT 4;5;6/4
|RUN
 1  2  3  4  5  6  7  8  9
 1           2
 3           4
 5           6
 1
 2
 3

 1  2  3  4  5  1.5
```

PRINT USING


Format

Print to the Screen
   PRINT USING ;§format-string§;[:print-list]
              ;§line-number §;
Print to a File (or Device)
   PRINT #file-number[,REC record-number],USING ;§format-string§;[:print-list]
                                                ;§line-number §;


Purpose

The PRINT USING instruction enables you to define specific formats for numbers
and strings you print.

You can use PRINT USING as either a program statement or a command.


Cross Reference

IMAGE, PRINT

---------------------------------------------------------------------------

   !o!    format-string--The format-string specifies the print format.
          The format-string is a string expression; if you use a string
          constant you must enclose it in quotation marks.  See IMAGE for
          an explanation of format-strings.

   !o!    line-number--You can optionally define a format-string in an
          IMAGE statement, as specified by the line-number.

   !o!    print-list--See PRINT for an explanation of the print-list and
          print options.

The PRINT USING instruction is identical to the PRINT instruction with the
addition of the USING option, except that:
   !o!      You cannot use the TAB function.

   !o!      You cannot use any print separator other than a comma (,),
            except that the print-list can end with a semicolon (;).

   !o!      If you use PRINT USING to print to a file, the file must have
            been opened in DISPLAY format.

EXAMPLES
|100 PRINT USING "###.##":32
.5

Prints 32.50.


|100 PRINT USING "THE ANSWER
 IS ###.#":123.98

Prints THE ANSWER IS 124.0.


|100 PRINT USING 185:37.4,-8
6.2
.
.
.
|185 IMAGE ###.#

Prints the values of 37.4 and -86.2 using the IMAGE statement in line 185.

# RANDOMIZE

## Format

RANDOMIZE [seed]

## Purpose

The RANDOMIZE instruction varies the sequence of pseudo-random numbers generated by the RND function.

You can use RANDOMIZE as either a program statement or a command.

## Cross Reference

RND

------------------------------------------------------------------------

!o!    seed--The optional seed is a numeric expression whose value specifies the random number sequence to be generated by RND functions.  The first two bytes of the internal representation of the value of the seed determine the random number sequence generated by RND.  If the first two bytes of the seed are identical each time you run your program, the same random number sequence is generated; if the first two bytes are not equal, a different sequence is generated.  If you do not enter a seed, a different and unpredictable sequence of random numbers is generated by RND each time you run your program.

## PROGRAM
The following program illustrates a use of the RANDOMIZE statement.  It accepts a value for the seed and prints the first 10 values obtained using the RND function.

```
|100 CALL CLEAR
|110 INPUT "SEED: ":S
|120 RANDOMIZE S
|130 FOR A=1 TO 10::PRINT A;
RND::NEXT A::PRINT
|140 GOTO 110
```
       (Press CLEAR to stop the program.)

READ

Format

READ variable-list

Purpose

The READ statement enables you to assign constants (stored within your program in DATA statements) to variables.

Cross Reference

DATA, RESTORE

---

!o!    variable-list--The variable-list, consisting of one or more
       variables separated by commas, specifies the numeric and/or
       string variables that are to be assigned values. When a READ
       statement is executed, the variables in its variable-list are
       assigned values from the data-list of a DATA statement. Unless
       you use a RESTORE statement to specify otherwise, DATA
       statements are read in ascending line-number order.

If a data-list does not contain enough values to assign to all the variables,
the READ statement assigns values from subsequent DATA statements until all
the variables have been assigned a value. If there are no more DATA
statements, a program error occurs and the message Data error in line-number
is displayed.

If a numeric variable is specified in the variable-list, a numeric constant
must be in the corresponding position in the data-list of a DATA statement.
If a string variable is specified in the variable-list, either a string or a
numeric constant can be in the corresponding position in the DATA statement.

See the DATA statement for examples.

REAL

Format

REAL numeric-variable-list
REAL ALL

Purpose

The REAL instruction enables you to declare the data/type of specified numeric variables as REAL.

REAL variables have a greater range of values than do INTEGER variables and can contain decimal portions.

You can use REAL as either a program statement or a command.

Cross Reference

DEF, DIM, INTEGER, OPTION BASE, SUB

-------------------------------------------------------------------------

!o!    numeric-variable-list--The numeric-variable-list consists of one
       or more numeric variables separated by commas.  The variables
       are all assigned the REAL data-type.  A REAL statement with a
       numeric-variable-list must have a line number lower than any
       program reference to any variable in that list.

!o!    ALL--If you enter the ALL option, all numeric variables in your
       program are assigned the REAL data-type unless specifically
       declared as INTEGER.  A REAL statement with the ALL option must
       have a line number lower than any program reference to any
       numeric variable or array.

                                is used as a
when   A REAL ALL command does not affect any variables unless they
       follow it on a multiple-statement line.

       A REAL ALL statement in your main program does not affect the
       data-type of a numeric variable in a subprogram.

A numeric variable of the REAL data-type can be any number that can be
expressed by the computer.

If you do not specify the data-type of a numeric variable, it is assigned the
REAL data-type (unless your program includes an INTEGER ALL statement or
defines the specific variable as an integer).

REAL statements are evaluated during pre-scan, and are not executed.

You can also declare REAL variables by using the data-type option in the DEF,
DIM, and SUB statements.

## EXAMPLES
TREAL A

As a command, specifies that the variable A is a real number.

|100 REAL ALL

As a statement, specifies that all numeric variables in the program are real numbers, unless specifically declared as INTEGER.

|100 REAL X(20)

Reserves space in memory for 21 real number elements of array X if it is not preceded by an OPTION BASE 1 statement.

REC function--Record Number

## Format

REC(file-number)

## Type

INTEGER

## Purpose

The REC function returns a record number reflecting the position of the next record in the specified file.

------------------------------------------------------------------

!o!    file-number--The file-number is a numeric expression whose value
       specifies the number of the file as assigned in its OPEN
       instruction.

The REC function returns the number of the record in the specified file that is to be accessed by the next PRINT, INPUT, or LINPUT instruction (the next sequential record).  (REC always treats a file as if it were being accessed sequentially, even if it has been opened for relative access.)

The records in a file are numbered sequentially starting with zero.

## EXAMPLE
1100 PRINT REC(4)

Prints the position of the next record in the file that was opened as number 4.

PROGRAM

The following program illustrates a use of the REC function.

```
|100 CALL CLEAR
|110 OPEN #1:"DSK1.RNDFILE",
RELATIVE,INTERNAL
|120 FOR A=0 TO 3
|130 PRINT #1:"THIS IS RECOR
D",A
|140 NEXT A
|150 RESTORE #1
|160 FOR A=0 TO 3
|170 PRINT REC(1)
|180 INPUT #1:A$,B
|190 PRINT A$;B
|200 NEXT A
|210 CLOSE #1
|RUN
  0
THIS IS RECORD 0
  1
THIS IS RECORD 1
  2
THIS IS RECORD 2
  3
THIS IS RECORD 3
```

Line 110 opens a file.

Lines 120 through 140 write four records on the file.

Line 150 resets the file to the beginning.

Lines 160 through 200 print the file position and read and print the values at that position.

Line 210 closes the file.

REM--Remark

Format

REM <u>remark</u>
!   <u>remark</u>

Purpose

The REM statement enables you to document your program by including
explanatory remarks within the program itself.
-----------------------------------------------------------------------
      !o!     <u>remark</u>--You can use any character in a <u>remark</u>.

The length of a REM statement is limited only by the length of a program
statement.

A REM statement encountered during program execution is ignored by the
computer.

Trailing Remarks

In addition to the REM statement, trailing remarks can be added to the ends of
lines in TI Extended BASIC II, allowing detailed internal documentation of
programs.  An exclamation mark (!) begins each trailing remark.

<u>EXAMPLE</u>
|100 REM BEGIN SUBROUTINE

Identifies a section beginning a subroutine.


|100 FOR X=1 TO 16 ! BEGIN L
OOP

Identifies a section beginning a FOR-NEXT loop.

RESEQUENCE

Format

```
;§RESEQUENCE§; [initial-line-number][,increment]
;§RES       §;
```

Purpose

The RESEQUENCE command assigns new line numbers to all lines in the program currently in memory..

-------------------------------------------------------------------------
!o!    initial-line-number--If you enter an initial-line-number, the first line number assigned is one you specify. If you do not specify an initial-line-number, the computer starts with line number 100.

!o!    increment--Succeeding line numbers are assigned by adding the value of the numeric expression increment to the previous line number. Note that to specify an increment only (without specifying an initial-line-number), you must precede the increment with a comma. The default increment is 10.

To ensure that your program continues to function properly, all line-number references within your program are changed to reflect the newly assigned line numbers. (Line numbers mentioned in REM statements are not affected.) If an invalid line-number reference (a reference to a line number that does not exist in your program) is encountered, the computer changes the line-number reference to 32767, without displaying any error message or warning.

If the values you enter for the initial-line-number and increment would have the effect of creating a line number greater than 32767, the message Bad line number is displayed and the program is not resequenced.

EXAMPLES
|RES

Resequences the lines of the program in memory to start with 100 and number by 10s.

|RES 1000

Resequences the lines of the program in memory to start with 1000 and number by 10s.

|RES 1000,15

Resequences the lines of the program in memory to start with 1000 and number by 15s.

|RES ,15

Resequences the lines of the program in memory to start with 100 and number by 15s.

RESTORE


Format

Restore Data
        RESTORE [line-number] .
Restore a File
        RESTORE #file-number[,REC record-number]


Purpose

The RESTORE instruction specifies either the DATA statement to be used with
the next READ statement or the record to be accessed by the next
file-processing instruction.


Cross Reference

DATA, INPUT, PRINT, READ

-----------------------------------------------------------------------

RESTORE with DATA and READ Statements

        !o!     line-number—If you enter a line-number, the next READ statement
                executed assigns values beginning from the data-list in the
                specified DATA statement.

                If the specified line-number is not the line-number of a DATA
                statement, the computer uses the first DATA statement with a
                line-number higher than the one you specified.

                If there is no higher numbered DATA statement, a program error
                occurs and the message Data error in line-number is displayed
                (the line-number is the line number of the READ statement that
                caused the error).

                If you do not enter a line-number or a file-number, the next
                READ statement executed assigns values beginning from the
                data-list of the first DATA statement in your program.

                If there are no DATA statements in your program, the message
                Data error in line-number is displayed.


RESTORE with a File

        !o!     file-number—If you enter a file-number, RESTORE repositions the
                specified file at its first record, record zero (unless you use
                the REC option). The file-number is a numeric expression whose
                value specifies the number of the file as assigned in its OPEN
                instruction.


Page 187

!o!    record-number--If you use the REC option, the record-number is a numeric expression specifying the number of the record at which you want to position the file.  The records in a file are numbered sequentially, starting with zero.  The REC option can be used only with  a file opened for RELATIVE access.

RESTORE terminates any print-or input-pending conditions.

EXAMPLES
T100 RESTORE

Sets the next DATA statement to be used to the first DATA statement in the program.

|100 RESTORE 130

Sets the next DATA statement to be used to the DATA statement at line 130 or, if line 130 is not a DATA statement, to the next DATA statement after line 130.

|100 RESTORE #1

Sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #1 to be the first record in the file.

|100 RESTORE #4,REC H5

Sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #4 to be record H5.

# RETURN

## Format

With GOSUB and ON GOSUB
        RETURN
With ON ERROR
        RETURN [;§NEXT          §;
                ;§line-number§;]

## Purpose

The RETURN statement causes program control to return to the main program from a subroutine called by a GOSUB, ON GOSUB, or ON ERROR statement.

## Cross Reference

GOSUB, ON GOSUB, ON ERROR

------------------------------------------------------------------------

## RETURN with GOSUB and ON GOSUB

When the computer encounters a RETURN statement in a subroutine called by a GOSUB or ON GOSUB statement, program control returns to the statement immediately following the GOSUB or ON GOSUB statement.

No options are allowed with a RETURN statement in a subroutine called by a GOSUB or ON GOSUB statement.

## RETURN with ON ERROR

The action taken by the computer when it encounters a RETURN statement in a subroutine called by an ON ERROR statement depends on the RETURN option.

> !o!    NEXT--If you specify the NEXT option, program control returns to the statement immediately following the statement that caused the error.

> !o!    line-number--If you specify a line-number, program control is transferred to the specified program statement.
>
>         If you do not specify an option, program control returns to the statement that caused the error.  The statement is re-executed.

RETURN "clears" the error, so that it can no longer be analyzed by the ERR subprogram.

## PROGRAM

The following program illustrates a use of RETURN as used with GOSUB.  The program figures interest on an amount of money put in savings.

```
100 CALL CLEAR
110 INPUT "AMOUNT DEPOSITED
: ":AMOUNT
120 INPUT "ANNUAL INTEREST
RATE: ":RATE
130 IF RATE 1 THEN RATE=RAT
E*100
140 PRINT "NUMBER OF TIMES
COMPOUNDED"
150 INPUT "ANNUALLY: ":COMP
160 INPUT "STARTING YEAR: "
:Y
170 INPUT "NUMBER OF YEARS:
 ":N
180 CALL CLEAR
190 FOR A=Y TO Y+N
200 GOSUB 240
210 PRINT A,INT(AMOUNT*100+
.5)/100
220 NEXT A
230 STOP
240 FOR B=1 TO COMP
250 AMOUNT=AMOUNT+AMOUNT*RA
TE/(COMP*100)
260 NEXT B
270 RETURN
```

RETURN (with ON ERROR)

PROGRAM

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

```
|100 CALL CLEAR
|110 A=1
|120 ON ERROR 160
|130 X=VAL("D")
|140 PRINT 140
|150 STOP
|160 REM ERROR HANDLING
|170 IF A|4 THEN 220
|180 A=A+1
|190 PRINT 190
|200 ON ERROR 160
|210 RETURN
|220 PRINT 220 :: RETURN NEX
T

|RUN

  190
  190
  190
  190
  220
  140
```

Line 120 causes an error to transfer control to line 160.  Line 130 causes an error.

Line 170 checks to see if the error has occurred four times and transfers control to 220 if it has.  Line 180 increments the error counter by one.  Line 190 prints 190.  Line 200 resets the error handling to transfer to line 160.  Line 210 returns to the line that caused the error and executes it again.

Line 220, which is executed only after the error has occurred four times, prints 220 and returns to the line following the line that caused the error.

Line 140, the next one after the one that causes the error, prints 140.

Also see the example of the ON ERROR statement.

RND function--Random Number

Format

RND

Type

REAL

Purpose

The RND function returns a pseudo-random number.

Cross Reference

RANDOMIZE

---

RND returns the next pseudo-random number in the current series of
pseudo-random numbers.  The number returned is always greater than or equal to
0 and less than 1.

The numbers returned by RND are called "pseudo-random" because they are not
generated strictly at random, but are generated as members of predefined
series.  You can use the RANDOMIZE instruction to make the numbers generated
by RND more random.

The same sequence of random numbers is generated by RND each time you run a
particular program unless the program includes a RANDOMIZE instruction.

EXAMPLES

```
100 COLOR16=INT(RND*16)+1
```

Sets COLOR16 equal to some number from 1 through 16.

```
100 VALUE=INT(RND*16)+10
```

Sets VALUE equal to some number from 10 through 25.

```
100 LL(8)=INT(RND*(B-A+1))+
A
```

Sets LL(8) equal to some number from A through B.

RPT$ function--Repeat String

Format

RPT$(string-expression,numeric-expression)

Type

String

Purpose

The RPT$ function returns a string consisting of a specified string repeated a specified number of times.

----------------------------------------------------------------------

!o!    string-expression--The string-expression specifies the string to be repeated.  If you use a string constant, it must be enclosed in quotation marks.

!o!    numeric-expression--The value of the numeric-expression specifies the number of repetitions of the string-expression.

If the length of the string-expression and the value of the numeric-expression would create a string longer than 4090 characters, the excess characters are discarded and the following message is displayed:

     * WARNING
       STRING TRUNCATED

EXAMPLES
|100 M$=RPT$("ABCD",4)

Sets M$ equal to "ABCDABCDABCDABCD."

|100 CALL CHAR(244,RPT$("000
0FFFF",8))

Defines characters 244 through 247 with the string
"0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF."

|100 PRINT USING RPT$("#",40
):X$

Prints the value of X$ using an image that consists of 40 number signs.

RUN

## Format

Execute Program in Memory
        RUN [line-number]
Execute Program on External Device
        RUN file-specification

## Purpose

The RUN instruction causes the computer either to execute the program currently in memory or to both load and execute a program from an external storage device.

You can use RUN as either a program statement or a command.

When you use RUN as a program statement, one program can start the execution of another program.  This enables you to divide a large program into smaller segments, each of which can be loaded into memory only as needed.

--------------------------------------------------------------------------

!o!     line-number--If you specify a line-number, your program starts
        running at the specified program line.

!o!     file-specification--If you enter a file-specification, your
        program is first loaded into memory from the specified external
        device, and then executed starting from the lowest-numbered line
        in the program.  The file-specification is a string expression;
        if you use a string constant, you must enclose it in quotation
        marks.  For more information see  "File Specifications," on page
        X.

If you do not enter either a line-number or a file-specification, the computer executes the program currently in memory starting with the lowest-numbered line in the program.

Before the program starts running, the computer:

!o!    Sets the values of all numeric variables to zero.

!o!    Sets the values of all string variables to null strings (strings containing no characters).

!o!    Closes all open files.

!o!    Restores the default character definitions of all characters.

!o!    Restores the default foreground color (black) and background color (transparent) to all characters.

!o!    Restores the default screen color (cyan).

!o!    Deletes all sprites.

!o!    Resets the sprite magnification level to 1.

!o!    Checks for certain program errors.

RUN does not affect the graphics mode, margin settings, graphics colors (see DCOLOR), or current position (see DRAWTO).

EXAMPLES
|RUN

Causes the computer to begin execution of the program in memory.

|RUN 200
|100 RUN 200

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

|RUN "DSK1.PRG3"
|100 RUN "DSK1.PRG3"

Causes the computer to load and begin execution of the program named PRG3 from the diskette in disk drive 1.

|100 A$="DSK1.MYFILE"
|110 RUN A$

Causes the computer to load and begin execution of the program named MYFILE from the diskette in disk drive 1.

## PROGRAM

The following program illustrates the use of the RUN command used as a statement. It creates a "menu" and lets the person using the program choose what other program he wishes to run. The other programs should RUN this program rather than ending in the usual way, so that the menu is given again after they are finished.

```
100 CALL CLEAR
110 PRINT "1 PROGRAM 1."
120 PRINT "2 PROGRAM 2."
130 PRINT "3 PROGRAM 3."
140 PRINT "4 END."
150 PRINT
160 INPUT "YOUR CHOICE: ":C
170 IF C=1 THEN RUN "DSK1.P
RG1"
180 IF C=2 THEN RUN "DSK1.P
RG2"
190 IF C=3 THEN RUN "DSK1.P
RG3"
200 IF C=4 THEN STOP
210 GOTO 100
```

SAVE

Format

SAVE file-specification[,;§MERGE    §;
                         ;§PROTECTED§;]


Purpose

The SAVE command copies the program in memory to an external storage device. When you are using SAVE, your program remains in memory, even if an error occurs.

The saved program can later be loaded back into memory with the OLD command.

Cross Reference

MERGE, OLD

---

!o!    file-specification--The file-specification names the program to
       be stored (see "File Specifications," on page X).  The
       file-specification, a string constant, can optionally be
       enclosed in quotation marks.

!o!    MERGE--To specify that your program is to be available for
       merging with other programs, use the MERGE option.  If you use
       the MERGE option, the program is stored as a SEQUENTIAL file in
       DISPLAY format with VARIABLE records (see OPEN); MERGE can be
       used only with devices that accept these options.

       For more information about using MERGE with a particular device,
       refer to the owner's manual that comes with that device.

       If you do not use the MERGE option, your program cannot later be
       merged with another program.

!o!    PROTECTED--If you use the PROTECTED option, you ensure that the
       program, when subsequently loaded with the OLD command, cannot
       be listed, edited, or saved.
                                        automatically
       A protected program starts executing by itself when it is
       loaded; when the program ends (either normally or because of an
       error) or stops at a breakpoint, it is erased from memory.  As
       the PROTECTED option is not reversible, it is recommended that
       you keep an unprotected version of the program.  If you also
       wish to protect a diskette-based program from being deleted, use
       the protect feature of the Disk Manager cartridge.


SAVE removes any breakpoints you have set in your program.

EXAMPLES
|SAVE DSK1.PRG1

Saves the program in memory on the diskette in disk drive 1 under the name
PRG1.


|SAVE DSK1.PRG1,PROTECTED

Saves the program in memory on the diskette in disk drive 1 under the name
PRG1.  The program may be loaded into memory, but it may not be edited,
listed, or resaved.


|SAVE DSK1.PRG1,MERGE

Saves the program in memory on the diskette in disk drive 1 under the name
PRG1.  The program may later be merged with a program in memory by using the
MERGE command.

SAY subprogram


**Format**

CALL SAY(word-string[,direct-string][,...])


**Purpose**

The SAY subprogram enables you to instruct the computer to produce speech.


**Cross Reference**

SPGET

------------------------------------------------------------------------

!o!   word-string--Word-string is a string expression whose value is
      any of the words or phrases in the computer's resident
      vocabulary.  If you use a string constant, you must enclose it
      in quotation marks.  Alphabetic characters must be upper-case.

      The computer substitutes "UHOH" for a word-string not in the
      vocabulary.

      A speech phrase (more than one word) must be enclosed in pound
      signs (#).  A speech phrase must be predefined; that is, it must
      be resident in the computer's vocabulary.

      A compound is a new word formed by combining two words already
      in the vocabulary.  For example, SOME+THING produces "something"
      and THERE+FOUR produces "therefore."  A compound must not be
      enclosed in pound signs.

      See Appendix L for a list of the computer's resident vocabulary.

!o!   direct-string--Direct-string is a string expression whose value
      is the computer's internal representation of a word or phrase.
      You can use or modify a direct-string returned by the SPGET
      subprogram.

See Appendix M for information on adding suffixes to direct-strings.
You can specify multiple word-strings and direct-strings by alternating them.
To specify two consecutive word-strings or direct-strings, enter an extra
comma as a separator between them.


**EXAMPLES**
TIOO CALL SAY("HELLO, HOW AR
E YOU")

Causes the computer to say "Hello, how are you."

|CALL SAY(,A$,,B$)

Causes the computer to say the the words indicated by A$ and B$, which must have been returned by SPGET.

## PROGRAM
The following program illustrates using CALL SAY with a word-string and three direct-strings.

```
|100 CALL SPGET("HOW",X$)
|110 CALL SPGET("ARE",Y$)
|120 CALL SPGET("YOU",Z$)
|130 CALL SAY("HELLO",X$,,Y$
,,Z$)
```

SCREEN subprogram

**Format**
CALL SCREEN(background-color[,foreground-color])

**Purpose**

The SCREEN subprogram enables you to change the screen color.  The screen color is the color of the border and the color displayed when transparent is specified as the foreground-or background-color of a character, pixel, or block.

In Text Mode, SCREEN enables you to change the color of the displayed characters, as well as the color of the screen.

**Cross Reference**

COLOR, DCOLOR, GRAPHICS

--------------------------------------------------------------------

!o!    background-color--Background-color is a numeric expression whose value specifies a screen color from among the 16 available colors.

!o!    foreground-color--In Text Mode, foreground-color is a numeric expression whose value specifies a color from among the 16 available colors, representing the foreground-color of all 256 characters.

       If you specify a foreground-color and the computer is not in Text Mode, it has no effect.  If the computer is in Text Mode and you do not specify a foreground-color, the foreground-color remains unchanged.

When you enter TI Extended BASIC II, the background-color is cyan and the foreground-color is black.  When your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode, the default colors are restored.

The codes for the available colors are listed below and in Appendix J.

| CODE | COLOR |
|------|-------|
| 1 | Transparent |
| 2 | Black |
| 3 | Medium Green |
| 4 | Light Green |
| 5 | Dark Blue |
| 6 | Light Blue |
| 7 | Dark Red |
| 8 | Cyan |
| 9 | Medium Red |
| 10 | Light Red |
| 11 | Dark Yellow |
| 12 | Light Yellow |
| 13 | Dark Green |
| 14 | Magenta |
| 15 | Gray |
| 16 | White |

EXAMPLES
]100 CALL SCREEN(8)

Changes the screen to cyan, which is the standard screen color.


|100 CALL SCREEN(2)

Changes the screen to black.


PROGRAM
The following program uses CALL SCREEN with CALL VCHAR and PRINT in the Text
Mode to change the color of a character.

```
|100 CALL CLEAR
|110 CALL GRAPHICS(2)
|120 CALL VCHAR(12,12,33,3)
|130 CALL SCREEN(6,16)
|140 PRINT "DARK BLUE SCREE
N WITH WHITE LETTERS"
|150 GOTO 150
```
        (Press CLEAR to stop the program.)

Line 130 changes the screen to dark blue and the characters to white.

SEG$ function--String Segment

Format

SEG$(string-expression,start-position,length)

Type

String

Purpose

The SEG$ function returns a specified substring (segment of a string).

-----------------------------------------------------------------

!o!    string-expression--The string-expression specifies the string of
       which you want to specify a substring.  If you use a string
       constant, it must be enclosed in quotation marks.

!o!    start-position--The start-position is a numeric expression whose
       value specifies the character position in the string-expression
       where the substring begins.  The value of the start-position
       must be greater than zero.

!o!    length--The length is a numeric expression whose value specifies the
       length of the substring.

If the start-position is greater than the length of the string-expression, or
if the length is zero, SEG$ returns a null string.

If the specified length is greater than the remaining length of the
string-expression (starting from the specified start-position), SEG$ returns a
substring consisting of all the characters in the string-expression starting
from the start-position to the end of the string-expression.


EXAMPLES
1100 X$=SEG$("FIRSTNAME LAST
NAME",1,9)

Sets X$ equal to FIRSTNAME.


1100 Y$=SEG$("FIRSTNAME LAST
NAME",11,8)

Sets Y$ equal to LASTNAME.

|100 Z$=SEG$("FIRSTNAME LAST
NAME",10,1)

Sets Z$ equal to " ".


|100 PRINT SEG$(A$,B,C)

Prints the substring of A$ starting at the character at position B and
extending for C characters.

SGN function--Signum (Sign)

Format

SGN(numeric-expression)

Type

INTEGER

Purpose

The SGN function returns a number indicating the algebraic sign of the value of the numeric-expression.

------------------------------------------------------------------------

!o!     numeric-expression--If the value of the numeric-expression is negative, SGN returns a -1.

If the value of the numeric-expression is zero, SGN returns a 0.

If the value of the numeric-expression is positive, SGN returns a (+)1.

EXAMPLES
|100 IF SGN(X2)=1 THEN 300 ELSE 400

Transfers control to line 300 if X2 is positive and to line 400 if X2 is zero or negative.

|100 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.

SIN function--Sine

**Format**

SIN(<u>numeric-expression</u>)

**Type**

REAL

**Purpose**

The SIN function returns the sine of the angle whose measurement in radians is the value of the <u>numeric-expression</u>.

**Cross Reference**

ATN, COS, TAN

------------------------------------------------------------------------

> !o!     <u>numeric-expression</u>--The value of the <u>numeric-expression</u> cannot
> be less than -1.5707963267944E10 or greater than
> 1.5707963267944E10.

To convert the measure of an angle from degrees to radians, multiply by PI/180.

**PROGRAM**

The following program gives the sine for each of several angles.

```
|100 A=.5235987755982
|110 B=30
|120 C=45*PI/180
|130 PRINT SIN(A);SIN(B)
|140 PRINT SIN(B*PI/180)
|150 PRINT SIN(C)
|RUN
 .5  -.9880316241
 .5
 .7071067812
```

SOUND subprogram

Format

CALL SOUND(duration,frequency1,volume1[,frequency2,volume2]
        [,frequency3,volume3][,frequency4,volume4]

Purpose

The SOUND subprogram enables you to instruct the computer to produce musical
tones or noise.

-----------------------------------------------------------------

The computer contains three music generators and one noise generator, enabling
you to create up to four different sounds at once.  You can specify the
frequency and volume of each sound independently.

!o!     duration--Duration is a numeric expression whose absolute value
        specifies the length of the sound in milliseconds (thousandths
        of seconds).  Duration can have an absolute value from 1 to
        4250.  (A value of 1000 will produce a sound for one second.)

        The actual duration produced by the computer may vary by as much
        as one sixtieth (1/60) of a second from the value you specify.

        You can enter only one duration, which applies to all specified
        sounds (music and noise).

!o!     frequency--Frequency is a numeric expression that has different
        meanings depending on whether you use it to specify one of the
        music generators or the noise generator.

        You must enter at least one frequency.

        Music--The frequency of a music generator specifies the
        frequency of the tone in Hertz (cycles per second).  The
        acceptable values range from 110 to 44733; the upper limit
        exceeds the range of human hearing.

        The actual frequency produced by the computer may vary by as
        much as ten percent from the value you specify.

        See Appendix H for the frequencies of some commonly used tones.

Noise--The frequency of the noise generator has a value from -1 to -8, specifying the type of noise produced.

The frequencies from -1 to -3 produce different types of periodic noise.  A frequency of -4 produces a periodic noise that varies depending on the frequency value of the third music generator.

The frequencies from -5 to -7 produce different types of white noise.  A frequency of -8 produces a white noise that varies depending on the frequency value of the third music generator.

!o!  volume--Volume is a numeric expression whose value is inversely proportional to the loudness of the sound.

You must enter at least one volume.

The volume can be from 0 to 30.  Zero is the maximum volume and 30 is silence.

If you call SOUND while the computer is still producing the tones specified in a previous call to the SOUND subprogram, the result depends on the algebraic sign of the duration of the previous call to SOUND.  If the duration was positive, the new sound does not begin until the old sound is complete.  If the duration was negative, the new sound begins immediately, interrupting the old sound.

EXAMPLES
```
1100 CALL SOUND(1000,110,0)
```

Plays A below low C loudly for one second.

```
|100 CALL SOUND(500,110,0,13
1,0,196,3)
```

Plays A below low C and low C loudly, and G below C not as loudly, all for half a second.

```
|100 CALL SOUND(4250,-8,0)
```

Plays loud white noise for 4.250 seconds.

```
|100 CALL SOUND(DUR,TONE,VOL
)
```

Plays the tone indicated by TONE for a duration indicated by DUR, at a volume indicated by VOL.

**PROGRAM**

The following program plays the 13 notes of the first octave that is available on the computer.

```
|100 X=2^(1/12)
|110 FOR A=1 TO 13
|120 CALL SOUND(100,110*X^A,
0)
|130 NEXT A
```

SPGET subprogram--Get Speech

## Format

CALL SPGET(<u>word-string,string-variable</u>[,...])

## Purpose

The SPGET subprogram enables you to assign the computer's internal representation of a speech word to a variable.

SPGET is especially useful if you want to add a suffix to a word in the computer's resident vocabulary.

## Cross Reference

SAY

---

!o!     <u>word-string</u>--<u>Word-string</u> is a string expression whose value is any of the words or phrases in the computer's resident vocabulary.  If you use a string constant, you must enclose it in quotes.

The computer substitutes "UHOH" for a <u>word-string</u> not in the vocabulary.

A speech phrase (more than one word) must be enclosed in pound signs (#).

See Appendix L for a list of the computer's resident vocabulary.

!o!     <u>string-variable</u>--The internal representation of the <u>word-string</u> (the <u>direct-string</u>) is returned in the <u>string-variable</u>.  See Appendix M for information on adding suffixes to <u>direct-strings</u>.

You can specify multiple <u>word-strings</u> and <u>direct-strings</u> by alternating them.

## PROGRAM
The following program illustrates using CALL SPGET.

```
|100 CALL SPGET("#TEXAS INST
RUMENTS#",X$)
|110 CALL SPGET("COMPUTER",Y
$)
|120 CALL SAY("I AM A",X$,,Y
$)
```

SPRITE subprogram


**Format**

CALL SPRITE(#sprite-number,character-code,foreground-color,pixel-row,
     pixel-column[,vertical-velocity,horizontal-velocity][,...])


**Purpose**

The SPRITE subprogram enables you to create sprites.


**Cross Reference**

CHAR, COINC, COLOR, DELSPRITE, DISTANCE, GRAPHICS, LOCATE, MAGNIFY, MOTION,
PATTERN, POSITION, SCREEN

---

Sprites are graphics that can be assigned any valid color and placed anywhere
on the screen.  Sprites treat the screen as a grid 256 pixels high and 256
pixels wide.  However, only the first 192 pixels are visible on the screen.

Sprites can be set in motion in any direction at a variety of speeds.  A
sprite continues its motion until it is specifically changed by the program or
until program execution stops.  Because sprites move from pixel to pixel,
their motion can be smoother than that of characters, which can be moved only
one character position (8 pixels) at a time.

You can create up to 32 sprites in all modes except Text Mode, which does not
allow sprites (the SPRITE subprogram has no effect in Text Mode).

Sprites "pass over" characters on the screen.  When two or more sprites are
coincident (occupying the same screen pixel position), the sprite with the
lowest sprite-number covers the other sprite(s).

At any given time, only four sprites can be on the same horizontal pixel-row.
When five or more sprites are on the same pixel-row, that row of pixels in the
sprite(s) with the highest sprite-number(s) disappears.

You can use the DELSPRITE subprogram to delete one or more sprites.  All
sprites are deleted when your program ends (either normally or because of an
error), stops at a breakpoint, or changes graphics mode.

Sprite Specifications

!o!　　sprite-number--The sprite-number is a numeric expression with a
　　　　value from 1 to 32.  If you specify the value of a previously
　　　　defined sprite, the old sprite is replaced by the new sprite.
　　　　If the old sprite had a vertical- or horizontal-velocity and you
　　　　do not specify a new velocity, the new sprite retains the old
　　　　velocity.

!o!　　character-code--Character-code is a numeric expression with a
　　　　value from 0-255, specifying the character that defines the
　　　　sprite pattern.

　　　　If you use the MAGNIFY subprogram to change to double-sized
　　　　sprites, the sprite definition includes the character specified
　　　　by the character-code and three additional characters (see
　　　　MAGNIFY).

　　　　Once defined by the SPRITE subprogram, the character-code of a
　　　　sprite can be changed by the PATTERN subprogram.

!o!　　foreground-color--The foreground-color is a numeric expression
　　　　with a value from 1 to 16, specifying one of the 16 available
　　　　colors.  Once defined by the SPRITE subprogram, the
　　　　foreground-color of a sprite can be changed by the COLOR
　　　　subprogram.

　　　　The background-color of a sprite is always transparent.

!o!　　pixel-row, pixel-column--The pixel-row and pixel-column are
　　　　numeric expressions whose values specify the screen pixel
　　　　position of the pixel at the upper-left corner of the sprite.

　　　　Once defined by the SPRITE subprogram, the pixel-row and
　　　　pixel-column of a sprite can be changed by the LOCATE
　　　　subprogram, and the current pixel-row and pixel-column of a
　　　　sprite can be ascertained by the POSITION subprogram.  Also, the
　　　　distance between sprites or between a sprite and a specified
　　　　screen pixel can be ascertained by the DISTANCE subprogram, and
　　　　the COINC subprogram can be used to ascertain whether sprites
　　　　are coincident with each other or with a specified screen pixel.

Sprite Motion

!o!    vertical-velocity, horizontal-velocity--The optional vertical-
       and horizontal-velocity are numeric expressions with values from
       -128 to 127. If both values are zero, the sprite is
       stationary. The speed of a sprite is in direct linear
       proportion to the absolute value of the specified velocity.

       A positive vertical-velocity causes the sprite to move toward
       the top of the screen; a negative vertical-velocity causes the
       sprite to move toward the bottom of the screen.

       A positive horizontal-velocity causes the sprite to move to the
       right; a negative horizontal-velocity causes the sprite to move
       to the left.

       If neither the vertical- nor horizontal-velocity are zero, the
       sprite moves at an angle, in a direction and at a speed
       determined by the velocity values.

The velocity of a sprite can be changed by the MOTION subprogram.

When a moving sprite reaches an edge of the screen, it disappears. The sprite
reappears in the corresponding position at the opposite edge of the screen.

The motion of a sprite may be affected by the computer's internal processing
and by input to and output from external devices.

PROGRAMS
The following three programs show some possible uses of sprites.

```
|100 CALL CLEAR
|110 CALL CHAR(244,"FFFFFFFF
FFFFFFFF")
|120 CALL CHAR(246,"183C7EFF
FF7E3C18")
|130 CALL CHAR(248,"F00FF00F
F00FF00F")
|140 CALL SPRITE(#1,244,5,92
,124,#2,248,7,1,1)
|150 CALL SPRITE(#28,33,16,1
2,48,1,1)
|160 CALL SPRITE(#15,246,14,
1,1,127,-128)
|170 GOTO 170
```
          (Press CLEAR to stop the program.)

Line 140 creates a dark blue sprite in the center of the screen and a red
striped sprite in the upper-right corner of the screen. Line 150 creates a
white sprite near the upper-left corner of the screen and starts it moving
slowly at a 45-degree angle down and to the right. The sprite is an
exclamation point.

Line 160 creates a dark red sprite at the upper-right corner of the screen and
starts it moving very fast at a 45 degree angle down and to the left.

The following program makes a rather spectacular use of sprites.

```
|100 CALL CLEAR
|110 CALL CHAR(244,"0008081C
7F1C0808")
|120 RANDOMIZE
|130 CALL SCREEN(2)
|140 FOR A=1 TO 28
|150 CALL SPRITE(#A,244,INT(
A/3)+3,92,124,A*INT(RND*4.5)
-2.25+A/2*SGN(RND-.5),A*INT(
RND*4.5)-2.25+A/2*SGN(RND-.5
))
|160 NEXT A
|170 GOTO 140
        (Press CLEAR to stop the program.)
```

Line 110 defines character 244.

Line 150 defines the sprites, 28 in all.  The sprite-number is the current
value of A.  The character-value is 244.  The sprite-color is INT(A/3)+3.  The
starting dot-row and dot-column are 92 and 124, the center of the screen.  The
row- and column-velocities are chosen randomly using the value of·
A*INT(RND*4.5) -2.25+A/2*SGN(RND-.5).

Line 170 causes the sequence to repeat.


The following program uses all the subprograms that relate to sprites except
for COLOR.  They are CHAR, COINC, DELSPRITE, LOCATE, MAGNIFY, MOTION, PATTERN,
POSITION, and SPRITE.

The program creates two double-sized magnified sprites in the shapes of two
people walking along a floor.  There is a barrier that one of them passes
through and the other jumps through.  The one that jumps through goes a little
faster after each jump, eventually catching the other one.  When this happens,
they each become double-sized, unmagnified sprites and continue walking.  When
they meet the second time, the one that has been going faster disappears and
the other continues walking.

```
|100 CALL CLEAR
|110 S1$="0103030103030303030303030303030380C0C080C0C0C0C0C0C0C0C0C0C0C0C0E0"
|120 S2$="0103030103070F1B1B03030306C0C0E80C0C080C0E0F0D8CCC0C0C060303038"
|130 COUNT=0
|140 CALL CHAR(244,S1$)
|150 CALL CHAR(248,S2$)
|160 CALL SCREEN(14)
|170 CALL COLOR(14,13,13)
|180 FOR A=19 TO 24
|190 CALL HCHAR(A,1,136,32)
|200 NEXT A
|210 CALL COLOR(13,15,15)
|220 CALL VCHAR(14,22,128,6)
|230 CALL VCHAR(14,23,128,6)
|240 CALL VCHAR(14,24,128,6)
|250 CALL SPRITE(#1,244,5,113,129,#2,244,7,113,9)
|260 CALL MAGNIFY(4)
|270 XDIR=4
|280 PAT=2
|290 CALL MOTION(#1,0,XDIR,#2,0,4)
|300 CALL PATTERN(#1,246+PAT,#2,246-PAT)
|310 PAT=-PAT
|320 CALL COINC(ALL,CO)
|330 IF CO |0 THEN 370
|340 CALL POSITION(#1,YPOS1,XPOS1)
|350 IF XPOS1|136 AND XPOS1 192 THEN 470
|360 GOTO 300
|370 REM COINCIDENCE
|380 CALL MOTION(#1,0,0,#2,0,0)
|390 CALL PATTERN(#1,244,#2,244)
|400 IF COUNT|0 THEN 540
|410 COUNT=COUNT+1
|420 CALL POSITION(#1,YPOS1,XPOS1,#2,YPOS2,XPOS2)
```

```
|430 CALL MAGNIFY(3)
|440 CALL LOCATE(#1,YPOS1+16
,XPOS1+8,#2,YPOS2+16,XPOS2)
|450 CALL MOTION(#1,0,XDIR,#
2,0,4) .
|460 GOTO 340
|470 REM #1 HIT WALL
|480 CALL MOTION(#1,0,0)
|490 CALL POSITION(#1,YPOS1,
XPOS1)
|500 CALL LOCATE(#1,YPOS1,19
3)
|510 XDIR=XDIR+1
|520 CALL MOTION(#1,0,XDIR)
|530 GOTO 300
|540 REM SECOND COINCIDENCE
|550 FOR DELAY=1 TO 1000 ::
NEXT DELAY
|560 CALL MOTION(#2,0,4)
|570 CALL DELSPRITE(#1)
|580 FOR STEP1=1 TO 20
|590 CALL PATTERN(#2,248)
|600 FOR DELAY=1 TO 40 :: NE
XT DELAY
|610 CALL PATTERN(#2,244)
|620 FOR DELAY=1 TO 40 :: NE
XT DELAY
|630 NEXT STEP1
|640 CALL CLEAR
```

Lines 110, 120, 140, 150, 250, and 260 define the sprites.

Line 130 sets the meeting counter to zero.

Lines 170 through 200 build the floor.

Lines 210 through 240 build the barrier.

Line 270 sets the starting speed of the sprite that will speed up.

Line 290 sets the sprites in motion.

Line 300 creates the illusion of walking.

Line 320 checks to see if the sprites have met.  Line 330 transfers control if the sprites have met.  Lines 340 and 350 check to see if the sprite has reached the barrier and transfer control if it has.

Line 360 loops back to continue the walk.  Lines 370 through 460 handle the sprites running into each other.  Lines 380 and 390 stop them.

Line 400 checks to see if it is the first meeting.  Line 410 increments the meeting counter.  Line 420 finds their position.

*sprites'*

Line 430 makes them smaller.  Line 440 puts them on the floor and moves the fast one slightly ahead.  Line 450 starts them moving again.

Lines 470 through 530 handle the fast sprite jumping through the barrier.  Line 480 stops it.  Line 490 finds where it is.

Line 500 puts it at the new location beyond the barrier.  Lines 510 and 520 start it moving again, a little faster.

Lines 540 through 640 handle the second meeting.

Line 560 starts the slow sprite moving.  Line 570 deletes the fast sprite.

Lines 580 through 630 make the slow sprite walk 20 steps.

SQR function--Square Root

Format

SQR(numeric-expression)

Type

REAL

Purpose

The SQR function returns the positive square root of the value of the numeric-expression.

------------------------------------------------------------------------
!o!    numeric-expression--The value of the numeric-expression cannot
       be negative.

EXAMPLES
1100 PRINT SQR(4)

Prints 2.

1100 X=SQR(2.57E5)

Sets X equal to the square root of 257,000, which is 506.9516742255.

STOP

Format

STOP

Purpose

The STOP statement stops the execution of your program.

Cross Reference

END

---

When your program encounters a STOP statement, the computer performs the following operations:

- !o!  It closes all open files.

- !o!  If the computer is in Pattern or Text Mode or one of the Split-Screen Modes, it restores the default character definitions of all characters.

- !o!  If the computer is in High-Resolution or Multicolor Mode, it restores the default character definitions of all characters and restores the default graphics mode (Pattern) and margin settings (2, 2, 0, 0).

- !o!  Restores the default foreground color (black) and background color (transparent) to all characters.

- !o!  Restores the default screen color (cyan).

- !o!  Deletes all sprites.

- !o!  Resets the sprite magnification level to 1.

The graphics colors (see DCOLOR) and current position (see DRAWTO) are not affected. If the computer is in Pattern or Text Mode or one of the Split-Screen Modes, the graphics mode and margin settings remain unchanged.

A STOP statement is not necessary to stop your program; the program automatically stops after the highest numbered line is executed.

STOP is frequently used before a subprogram that follows the main portion of a program, to ensure that the subprogram is not executed after the execution of the highest numbered line in the main program.

STOP can be used interchangeably with the END statement, except that you cannot use STOP after a subprogram.                    *to end*

## PROGRAM
The following program illustrates the use of the STOP statement.  The program adds the numbers from 1 to 100.

```
|100 CALL CLEAR
|110 TOT=0
|120 NUMB=1
|130 TOT=TOT+NUMB
|140 NUMB=NUMB+1
|150 IF NUMB|100 THEN PRINT
TOT::STOP
|160 GOTO 130
```

STR$ function--String-Number

Format

STR$(numeric-expression)

Type

String

Purpose

The STR$ function returns the string representation of the value of the numeric-expression.

STR$ enables you to use the string representation of the numeric-expression with an instruction that requires a string expression as a parameter.

STR$ is the inverse of the VAL function.

Cross Reference

VAL

---

STR$ removes leading and trailing spaces.

EXAMPLES
|100 NUM$=STR$(78.6)

Sets NUM$ equal to "78.6".

|100 LL$=STR$(3E15)

Sets LL$ equal to "3.E+15".

|100 X$=STR$(A*4)

representation of

Sets X$ equal to a string equal to whatever value is obtained when A is multiplied by 4.  For instance, if A is equal to -8, X$ is set equal to "-32".

SUB--Subprogram

Format

SUB subprogram-name[([data-type ]parameter[,...])]

Purpose

The SUB statement is the first statement in a subprogram.

You can use a subprogram to separate a group of statements from the main program. Subprograms are generally used to perform a specific operation several times in the same program or in different programs, or to isolate variables that are specific to the subprogram.

Cross Reference

CALL, SUBEND, SUBEXIT

--------------------------------------------------------------------

!o!    subprogram-name--Subprograms are accessed from your main program
       with a CALL statement. The subprogram-name in the SUB statement
       is the same name that you use in the CALL statement that
       transfers control to the subprogram.

       The maximum length of a subprogram-name is 15 characters.

       A user-written subprogram may have the same subprogram-name as a
       built-in subprogram. In such a case, a CALL statement will
       access the user-written subprogram instead of the built-in one.

!o!    parameter--You can use parameters to pass values to a
       subprogram. Parameters must be valid names of variables or
       arrays.

!o!    data-type--If a parameter is numeric, you can optionally specify
       its data-type (INTEGER or REAL). Numeric parameters are
       considered to be REAL unless you specifically declare them as
       INTEGER in the parameter list. An INTEGER ALL statement in your
       main program or in the subprogram itself does not affect the
       data-type of parameters.

SUBEND must be the last statement in a subprogram. When the computer     exec
encounters a SUBEND or a SUBEXIT statement in a subprogram, program control
returns to the statement immediately following the CALL statement that called
the subprogram.

It is recommended that you do not use any statement other than SUBEND or
SUBEXIT to leave a subprogram. If you use another statement to leave a
subprogram you may still be using variables local to the subprogram, which may
cause unexpected results.

Subprograms must have higher line numbers than any part of your main program. A SUB statement cannot be part of an IF THEN statement.

## Subprogram Variables

The variables used in a subprogram (other than those used as parameters) are local to the subprogram; that is, even if a variable in your main program has the same name as a variable in a subprogram, the value of that variable outside the subprogram is not affected by changes to its value in the subprogram.  If a subprogram is called more than once, any local variables used in the subprogram retain their values from one call to the next.

Numeric variables used in a subprogram are considered to be REAL unless you specifically declare them as INTEGER.  An INTEGER ALL statement in your main program does not affect the data-type of numeric variables in a subprogram.

An INTEGER ALL statement in a subprogram does not affect the data-type of numeric variables in your main program or in any other subprogram.

## Parameters

When your program executes a subprogram beginning with a SUB statement with parameters, the parameter values (constants or variables) are passed from the parameter-list of the CALL statement to the subprogram.  The parameter-list in the CALL statement must contain the same number of parameters as the SUB statement.  Values are passed in the order in which they are listed.

A numeric parameter must be passed a numeric value.  A string parameter must be passed a string value.

An array parameter must be passed an array.  A string-array parameter must be passed a string array; an INTEGER-array parameter must be passed an INTEGER array; a REAL-array parameter must be passed a REAL array.

To pass an entire array as one parameter, follow the array name with left and right parentheses.  If the array has more than one dimension, place one comma between the parentheses for each additional dimension.

Passing Parameters by Reference and Value

When a subprogram manipulates the value of a <u>parameter</u> passed to it, the new
<u>parameter</u> value may or may not be passed back to the main program.
When a <u>parameter</u> is passed to a subprogram "by reference," the new value is
passed back to the main program after the subprogram has executed.

When a <u>parameter</u> is passed to a subprogram "by value," the new value is not
passed back to the main program.

!o!     Variables, array elements, and arrays are normally passed by
        reference.  However, if a numeric variable or array element is
        of a different <u>data-type</u> in the main program than it is in the
        subprogram (INTEGER vs. REAL), the <u>parameter</u> is passed by
        value.  Note that if you pass an INTEGER variable to a numeric
        <u>parameter</u>, that subprogram <u>parameter</u> is considered to be REAL
        unless you specifically declare it as an INTEGER by using the
        <u>data-type</u> option in the <u>parameter</u> list of the SUB statement.
        Remember that if you want to pass an INTEGER by reference, you
        must declare the subprogram <u>parameter</u> as an INTEGER.

!o!     To specify that a variable or array element is to be passed by
        value rather than by reference, enclose it in parentheses in the
        CALL statement's <u>parameter-list</u>.  Note that this option is not
        available for arrays.

!o!     If you use an expression as a <u>parameter</u>, it is evaluated and
        passed by value.

EXAMPLES
|100 SUB MENU

Marks the beginning of a subprogram.  No parameters are passed or returned.


|100 SUB MENU(COUNT,CHOICE)

Marks the beginning of a subprogram.  The variables COUNT and CHOICE may be
used and/or have their values changed in the subprogram and returned to the
variables in the same position in the calling statement.


|100 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 may be used and/or have their values
changed in the subprogram and returned to the variables in the same position
in the calling statement.

## PROGRAM

The following program illustrates the use of SUB.  The subprogram MENU had been previously saved with the merge option.  It prints a menu and requests a choice.  The main program tells the subprogram how many choices there are and what the choices are.  It then uses the choice made in the subprogram to determine what program to run.

```
|100 CALL MENU(5,R)
|110 ON R GOTO 120,130,140,1
|50,160
|120 RUN "DSK1.PAYABLES"
|130 RUN "DSK1.RECEIVE"
|140 RUN "DSK1.PAYROLL"
|150 RUN "DSK1.INVENTORY"
|160 RUN "DSK1.LEDGER"
|170 DATA ACCOUNTS PAYABLE,A
CCOUNTS RECEIVABLE,PAYROLL,I
NVENTORY,GENERAL LEDGER
```

Beginning of subprogram MENU.

Note that this R is not the same as the R used in lines 100 and 110 in the main program.

```
|10000 SUB MENU(COUNT,CHOICE
)
|10010 CALL CLEAR
|10020 IF COUNT|22 THEN PRIN
T "TOO MANY ITEMS" :: CHOICE
=0 :: SUBEXIT
|10030 RESTORE
|10040 FOR R=1 TO COUNT
|10050 READ TEMP$
|10060 TEMP$=SEG$(TEMP$,1,25
)
|10070 DISPLAY AT(R,1):R;TEM
P$
|10080 NEXT R
|10090 DISPLAY AT(R+1,1):"YO
UR CHOICE: 1"
|10100 ACCEPT AT(R+1,14)BEEP
 VALIDATE(DIGIT)SIZE(-2):CHO
ICE
|10110 IF CHOICE 1 OR CHOICE
|COUNT THEN 10100
|10120 SUBEND
```

— reverse relational expressions

SUBEND--Subprogram End

Format

SUBEND

Purpose

The SUBEND statement marks the end of a subprogram.

Cross Reference

SUB, SUBEXIT

-----------------------------------------------------------------

SUBEND must be the last statement in a subprogram.  When the computer
encounters a SUBEND statement in a subprogram, program control returns to the
statement immediately following the CALL statement that called the subprogram.

It is recommended that you do not use any statement other than SUBEND or
SUBEXIT to leave a subprogram.  If you use another statement to leave a
subprogram you may still be using variables local to the subprogram, which may
cause unexpected results.

A SUBEND statement cannot be part of an IF THEN statement.

The only statements that can immediately follow a SUBEND statement are REM,
END, or the SUB statement for the next subprogram.

SUBEXIT--Subprogram Exit

Format

SUBEXIT

Purpose

The SUBEXIT statement enables you to leave a subprogram before the computer executes the SUBEND statement that ends the subprogram.

SUBEXIT enables you to have more than one exit from a subprogram.

Cross Reference

SUB, SUBEND

------------------------------------------------------------------------

When the computer encounters a SUBEXIT statement in a subprogram, program control returns to the statement immediately following the CALL statement that called the subprogram.

It is recommended that you do not use any statement other than SUBEND or SUBEXIT to leave a subprogram.  If you use another statement to leave a subprogram you may still be using variables local to the subprogram, which may cause unexpected results.

TAB function--Tabulate

Format

TAB(numeric-expression)

Purpose

The TAB function specifies the starting position of the next item to be printed by a PRINT or DISPLAY instruction.

Cross Reference

DISPLAY, PRINT

---

!o!    numeric-expression--The numeric-expression specifies the starting position of the next print item in a print-list of a PRINT or DISPLAY instruction.

If the value of the numeric-expression is not an integer, it is rounded to the nearest integer.  If the value of the numeric-expression is less than 1, it is replaced by 1.

If the value of the numeric-expression is greater than the record length of the screen or device, it is repeatedly reduced by the record length until it is less than or equal to the record length.  The record length of the screen is the width of the screen window defined by the margins.  For more information about the record length of a particular device, refer to the owner's manual that comes with that device.

TAB is relative to the left side of the screen, not the current screen window.

Because the TAB function itself is treated as a separate print item, it must be preceded and/or followed by a print separator (usually a semicolon), unless it is the only item in the print-list.

If the number of characters already printed in the current record is greater than or equal to the position indicated by the value of the numeric-expression, the print item following the TAB is printed in the next record, beginning in the position specified by the value of the numeric-expression.

TAB can be used to print to a device or file only if the device or file has been open in DISPLAY format.

TAB cannot be used with PRINT USING or DISPLAY USING.

EXAMPLES
|100 PRINT TAB(12);35

Prints the number 35 at the twelfth position.


|100 PRINT 356;TAB(18);"NAME
"

Prints 356 at the beginning of the line and NAME at the eighteenth position of
the line.


|100 PRINT "ABCDEFGHIJKLM";T
AB(5);"NOP"

Prints ABCDEFGHIJKLM at the beginning of the line and NOP at the fifth
position of the next line.


|100 DISPLAY AT(12,1):"NAME"
;TAB(15);"ADDRESS"

Displays NAME at the beginning of the twelfth line on the screen and ADDRESS
at the fifteenth position on the twelfth line of the screen.

TAN function—Tangent

Format

TAN(numeric-expression)

Type

REAL

Purpose

The TAN function returns the tangent of the angle whose measurement in radians is the value of the numeric-expression.

Cross Reference

ATN, COS, SIN

-------------------------------------------------------------------------

> !o!  ·  numeric-expression—The numeric-expression cannot be less than −1.57079632695l4El0 or greater than 1.57079632663741El0.

To convert the measure of an angle from degrees to radians, multiply by PI/180.

PROGRAM
The following program gives the tangent for each of of several angles.

```
|100 A=.78539816339731
|110 B=26.565051177
|120 C=45*PI/180
|130 PRINT TAN(A);TAN(B)
|140 PRINT TAN(B*PI/180)
|150 PRINT TAN(C)
|RUN
  1.  7.17470553
  .5
  1
```

TERMCHAR function--Termination Character

Format

TERMCHAR

Type

INTEGER

Purpose

The TERMCHAR function returns the character code of the key pressed to exit from the previously executed INPUT, ACCEPT, or LINPUT statement.

Cross Reference

ACCEPT, INPUT, LINPUT

--------------------------------------------------------------------------
In a program, the value returned by TERMCHAR depends on the key pressed to exit from the last instruction that accepted input from the keyboard.

| VALUE RETURNED | KEY |
|---|---|
| 1 | AID |
| 2 | CLEAR |
| 10 | DOWN ARROW |
| 11 | UP ARROW |
| 12 | PROC'D |
| 13 | ENTER |
| 14 | BEGIN |
| 15 | BACK |

If you use TERMCHAR as part of a command (unless it is preceded by ACCEPT, INPUT, or LINPUT), the value returned depends on the key pressed to enter the command (ENTER, UP ARROW, or DOWN ARROW).

Note that pressing CLEAR during keyboard input normally causes a break in the program.  However, If your program includes an ON BREAK NEXT statement, you can use CLEAR to exit from an input field.

PROGRAM
The following program illustrates a use of TERMCHAR.  The program displays
name, address, and city, state, and zip code information entered from the
keyboard.  Line 160 enables you correct errors in previously entered lines by
pressing UP ARROW.  This returns the cursor to the beginning of the line that
immediately precedes the one from which UP ARROW was entered.

```
|100 CALL CLEAR
|110 R=5::C=12
|120 DISPLAY AT(R,C-10):"NAM
E :"
|130 DISPLAY AT(R+1,C-10):"A
DDRESS:"
|140 DISPLAY AT(R+2,C-10):"C
,S,Z:"
|150 ACCEPT AT(R,C)SIZE(-20)
:A$(R)
|160 IF TERMCHAR=11 THEN R=R
-1 ELSE R=R+1
|170 IF R =7 THEN 150
|180 DISPLAY AT(20,1):A$(5):
A$(6):A$(7)
|190 GOTO 110
        (Press CLEAR to stop the program.)
```

TRACE

Format

TRACE

Purpose

The TRACE instruction causes the computer to display the line number of each
line in your program before it is executed.

TRACE enables you to see the order in which the computer performs statements
as it runs your program.  It is valuable as a debugging aid to help you find
errors (such as unwanted infinite loops) in your program.

You can use TRACE as either a program statement or a command.

Cross Reference

UNTRACE

--------------------------------------------------------------------------------
The effect of a TRACE instruction is cancelled when an UNTRACE instruction or
a NEW command is executed.

PROGRAMS
The following programs display a trace of the order of execution of the
program lines.
|100 FOR J=1 TO 3
|110 PRINT "WORD"
|120 NEXT J
|130 TRACE


|100 FOR J=1 TO 3
|110 PRINT "WORD"
|120 NEXT J
|TRACE

## UNBREAK

### Format

UNBREAK [line-number-list]

### Purpose

The UNBREAK instruction removes a breakpoint from each program statement you specify.

You can use UNBREAK as either a program statement or a command.

### Cross Reference

BREAK

---

!o!   line-number-list--The line-number-list consists of one or more
      line numbers separated by commas. When an UNBREAK instruction is
      executed, breakpoints are removed from the specified program
      lines.

      If you do not include a line-number-list, UNBREAK removes all
      breakpoints, except for a breakpoint that occurs when a BREAK
      statement with no line-number-list is encountered in a program.

If the line-number-list includes an invalid line number (0 or a value greater
than 32767), the message Bad line number is displayed. If the
line-number-list includes a fractional or negative line number, the message
Syntax error is displayed. In both cases, the UNBREAK instruction is ignored;
that is, breakpoints are not removed even at valid line numbers in the
line-number-list. If you were entering UNBREAK as a program statement, it is
not entered into your program.

If the line-number-list includes a line number that is valid (1-32767) but is
not the number of a line in your program, or a fractional number greater than
1, the message

   * WARNING
     LINE NOT FOUND

is displayed. (If you were entering UNBREAK as a program statement, the
line-number is included in the warning message.) A breakpoint is, however,
removed from any valid line in the line-number-list that precedes the line
number that caused the warning.

EXAMPLES
| UNBREAK
| 450 UNBREAK

Removes all breakpoints (except those resulting from a BREAK statement with no line-number-list).

| UNBREAK 100,130
| 350 UNBREAK 100,130

Removes the breakpoints from lines 100 and 130.

UNTRACE

Format

UNTRACE

Purpose

The UNTRACE instruction cancels the effect of a TRACE instruction.

You can use UNTRACE as either a program statement or a command.

Cross Reference

TRACE

-----------------------------------------------------------------

EXAMPLES
TUNTRACE
|450 UNTRACE

Removes the effect of TRACE.

VAL function--Value

Format

VAL(string-expression)

Type

REAL

Purpose

The VAL function returns the numeric value of the string-expression.

VAL enables you to use the numeric value of the string-expression with an instruction that requires a numeric expression as a parameter.

VAL is the inverse of the STR$ function.

Cross Reference

STR$

-------------------------------------------------------------------

!o!   string-expression--The string-expression must be a valid
      representation of a number.  The length of the string-expression
      must be greater than 0 and less than 4090.  If you use a string
      constant, it must be enclosed in quotation marks.

EXAMPLES
```
100 NUMB=VAL("78.6")
110 PRINT NUMB
```

Prints 78.6.

```
100 LL=VAL("3E15")
```

Sets LL equal to 3E+15, or 315.

VALHEX function--Value of Hexadecimal Number

Format

VALHEX(string-expression)

Type

INTEGER

Purpose

VALHEX returns the numeric value of the hexadecimal number represented by the string-expression.

----------------------------------------------------------------

!o!  string-expression--The string-expression specifies the hexadecimal (base 16) number to be converted to a decimal (base 10) number.  If you use a string constant, it must be enclosed in quotation marks.

The string-expression must contain only valid hexadecimal digits (0-9, A-F).  Alphabetic hexadecimal digits must be upper-case letters.  VALHEX can convert a hexadecimal number from one to four digits long.  If the length of the string-expression is greater than four, VALHEX uses only the last four characters.

VALHEX returns an integer greater than or equal to -32768 (hexadecimal 8000) and less than or equal to 32767 (hexadecimal 7FFF).

EXAMPLES
100 A=VALHEX("400A")

Sets A equal to 16394.

100 PRINT VALHEX("8200")

Prints -32256.

VCHAR subprogram--Vertical Character

Format

CALL VCHAR(row,column,character-code[,number-of-repetitions])

Purpose

The VCHAR subprogram enables you to place a character on the screen and repeat it vertically.

Cross Reference

DCOLOR, GCHAR, GRAPHICS, HCHAR

---

VCHAR is effective only in Pattern, Text, and High-Resolution Modes and in the graphics portion of the screen in the Split-Screen Modes. VCHAR has no effect in Multicolor Mode, and cannot access the text portion of the screen in the Split-Screen Modes.

!o!  row, column--Row and column are numeric expressions whose values specify the character position on the screen where the character is displayed.

The value of row must be greater than or equal to 1. In Pattern, Text, and High-Resolution Modes, row must be less than or equal to 24; in the Split-Screen Modes, row must be less than or equal to 16.

The value of column must be greater than or equal to 1. In Pattern, High-Resolution, and Split-Screen Modes, column must be less than or equal to 32; in Text Mode, column must be less than or equal to 40.

VCHAR is not affected by margin settings in Pattern and Text Modes. Row and column are relative to the upper-left corner of the screen, not to the corner of the window defined by the margins.

In the Split-Screen Modes, the row and column are relative to the graphics portion of the screen. The upper-left corner of the graphics portion of the screen is considered to be the intersection of row 1 and column 1.

!o!  character-code--The character-code is a numeric expression with a value from 0-255, specifying the number of the character. See Appendix A for a list of ASCII character codes.

!o!   number-of-repetitions--The optional number-of-repetitions is a
      numeric expression whose value specifies the number of times the
      character is repeated vertically.  If the repetitions extend past
      the end of a column, they continue from the first character of
      the next column.  If the repetitions extend past the end of the
      last column, they continue from the first character of the first
      column.

If you use VCHAR to display a character on the screen, and then later use
CHAR, or COLOR to change the appearance of that character, the result depends
on the graphics mode.

      !o!   In Pattern and Text Modes, the displayed character changes to the
            newly specified pattern and/or color(s).

      !o!   In High-Resolution and Split-Screen Modes, the displayed
            character remains unchanged.


EXAMPLES
100 CALL VCHAR(12,16,33)

Places character 33 (an exclamation point) in row 12, column 16.


100 CALL VCHAR(1,1,ASC("!")
,768)

Places an exclamation point in row 1, column 1, and repeats it 768 times,
filling the screen.


100 CALL VCHAR(R,C,K,T)

Places the character with an ASCII code of K in row R, column C, and repeats it
T times.

VERSION subprogram

Format

CALL VERSION(numeric-variable)

Purpose

The VERSION subprogram returns a value indicating the version of BASIC being used.

---

!o! numeric-variable--In TI Extended BASIC II, VERSION returns a value of 200 to the numeric-variable you specify.

EXAMPLE
1100 CALL VERSION(V)

Sets V equal to 200.