CHAPTER V
REFERENCE SECTION

This chapter is an alphabetical list of the CC-40 BASIC command, statement, and function keywords. Each keyword is explained in the following sections.

The Format section gives the complete syntax of the keyword, using the following conventions.

- KEYWORDS are capitalized.
- *Variables* are in italics.
- All parentheses are mandatory. Parentheses included with an optional element must be included when the optional element is used.
- Optional elements are enclosed in [brackets].
- Items that may be repeated are indicated by ellipses (...).
- Items representing alternative forms are presented one above the other and are enclosed in {braces}.

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

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

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

ABS

Format

ABS(*numeric-expression*)

Description

The ABS function gives the absolute value of *numeric-expression*. If *numeric-expression* is positive or zero, ABS returns the value of *numeric-expression*. If *numeric-expression* is negative, ABS returns the negative of the value. The result of ABS is always positive or zero.

Examples

```
370 PRINT ABS(42.3):PAUSE
```

Prints 42.3.

```
140 VV=ABS(-6.124)
```

Sets VV equal to 6.124.

ACCEPT

Format

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

Description

The ACCEPT statement suspends program execution until data is entered from the keyboard. The options available with ACCEPT make it more versatile for keyboard entry than the INPUT statement. ACCEPT can accept data at any display position, sound an audible tone (beep), erase all or part of the display, limit the number and type of characters accepted, and provide a default value for the input variable.

AT(*column*) positions the cursor and the beginning of the input field at the specified *column*, which must be from 1 to 31. If AT is omitted, input begins in column one unless a previous input/output statement left the cursor positioned in columns 2 through 31, in which case input is accepted at the cursor location.

SIZE(*numeric-expression*) allows up to the absolute value of *numeric-expression* characters to be input. If *numeric-expression* is positive, the input field is cleared before input is accepted. If *numeric-expression* is negative, the input field is not cleared, thus allowing a default value previously placed into the field by a DISPLAY or PRINT statement to be entered. If SIZE is omitted, the ACCEPT statement clears the display from the current cursor position to the end of the 80-column line. If SIZE is used, the cursor is left in the first position following the input field for subsequent input/output statements.

BEEP sounds a short tone for each BEEP in the statement, to indicate that the computer is ready to accept input.

ERASE ALL clears the entire display before accepting input, and positions the cursor to column one. If AT is used, the data is accepted starting at the position specified by *column*.

(continued)

ACCEPT

(continued)

VALIDATE(*data-type*) allows only certain characters to be entered from the keyboard. Note that default values are not validated. *Data-type* specifies which characters are acceptable.

Data-type can be a string expression which specifies the characters that are permitted. Only one string expression may be specified for *data-type*. The following can also be used as *data-types*. If more than one *data-type* is specified, a character from any of the types specified is acceptable.

ALPHA	permits all alphabetic characters.
UALPHA	permits only uppercase alphabetic characters.
DIGIT	permits 0 through 9.
NUMERIC	permits 0 through 9, ".", "+", "−", and "E".
ALPHANUM	permits all alphabetic characters and 0 through 9.
UALPHANUM	permits only uppercase alphabetic characters and 0 through 9.

NULL(*expression*) provides a default value to be assigned to the variable if **[ENTER]** is pressed with a blank (or null) input field.

During the execution of an ACCEPT statement, the following types of entries are also permitted.

- The **[FN]** key can be used to input keywords and user-assigned strings from the keyboard.
- A numeric expression can be entered if *variable* is numeric. The expression is evaluated and the result is assigned to *variable*.
- **[SHIFT] [ENTER]** can be pressed to cause the input data to be ignored, the value of *variable* to remain unchanged, and the program to proceed to the next statement. If NULL is included, it is also ignored.

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

(continued)

ACCEPT

(continued)

Cross Reference

INPUT, LINPUT

Examples

```
100 ACCEPT AT(3) ERASE ALL,T
```

Clears the display, accepts data starting in column 3, and places the data into variable T.

```
320 ACCEPT VALIDATE("yn") SIZE(1),A$
```

Accepts a one character field consisting of either y or n into the variable A$.

```
430 ACCEPT AT(3) SIZE(-5) BEEP VALIDATE(DIGIT,"+-"),X
```

Beeps, then accepts up to 5 characters for the variable X, starting at column 3. The input characters must consist of digits or the characters + or −. Because the SIZE specification is negative, the input field is not erased prior to accepting input.

```
570 ACCEPT NULL(PI),C
```

Accepts data for C. If no data has been entered when **[ENTER]** is pressed, the value of PI is stored in the variable C.

ACS

Format

ACS(*numeric-expression*)

Description

The ACS (arccosine) function calculates the angle whose cosine is *numeric-expression.* The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ACS function for the three angle settings is shown below.

Units	Range of Calculated Angles
Degrees	0 ≤ ACS(X) ≤ 180
Radians	0 ≤ ACS(X) ≤ PI
Grads	0 ≤ ACS(X) ≤ 200

Examples

```
100 DEG
```
Selects DEG angle setting.

```
110 PRINT ACS(1):PAUSE
```
Prints 0.

```
220 RAD
```
Selects RAD angle setting.

```
230 T=ACS(0.75)
```
Sets T equal to .72273424781339.

SUBPROGRAM

ADDMEM

Format

CALL ADDMEM

Description

The ADDMEM subprogram allows the Random Access Memory (RAM) contained in an installed *Memory Expansion* cartridge to be appended to the useable resident memory. The amount of memory added is described in appendix J.

CALL ADDMEM cannot be used in a program. The error message `No RAM in cartridge` is displayed if no *Memory Expansion* cartridge is installed when CALL ADDMEM is executed.

When the memory in a *Memory Expansion* cartridge has been appended to resident memory, the system is initialized if a loss of memory is detected (i.e. power is lost or the cartridge is removed) or the reset key is pressed.

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

Example

```
CALL ADDMEM
```
Allows the use of memory supplied by the installed *Memory Expansion* cartridge. See chapter 3 or appendix J for more information.

ASC

Format

ASC(*string-expression*)

Description

The ASC function returns the ASCII character code of the first character of *string-expression*. The message Bad argument is displayed if *string-expression* is a null string. A list of the ASCII codes is given in appendix D. The ASC function is the inverse of the CHR$ function.

Cross Reference

CHR$

Examples

```
100 PRINT ASC("A"):PAUSE
```
Prints 65.

```
130 B=ASC("1")
```
Sets B equal to 49.

```
790 DISPLAY ASC("HELLO"):PAUSE
```
Displays 72.

ASN

Format

ASN(*numeric-expression*)

Description

The ASN (arcsine) function calculates the angle whose sine is *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ASN function for the three angle settings is shown below.

Units	Range of Calculated Angles
Degrees	$-90 \leq ASN(X) \leq 90$
Radians	$-PI/2 \leq ASN(X) \leq PI/2$
Grads	$-100 \leq ASN(X) \leq 100$

Examples

```
140 DEG
```
Selects DEG angle setting.

```
150 PRINT ASN(1):PAUSE
```
Prints 90.

```
240 RAD
```
Selects RAD angle setting.

```
250 B=ASN(.9)
```
Sets B equal to 1.119769514999.

ATN

Format

ATN(*numeric-expression*)

Description

The ATN (arctangent) function calculates the angle whose tangent is *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. The range of values given by the ATN function for the three angle settings is shown below.

Units	Range of Calculated Angles
Degrees	$-90 < ATN(X) < 90$
Radians	$-PI/2 < ATN(X) < PI/2$
Grads	$-100 < ATN(X) < 100$

Examples

```
130 GRAD
```
Selects GRAD angle setting.

```
140 PRINT ATN(30):PAUSE
```
Prints 97.87871952.

```
810 RAD
```
Selects RAD angle setting.

```
820 Q=ATN(2.5)
```
Sets Q equal to 1.190289949683.

ATTACH

Format

ATTACH *sub-name1* [, *sub-name2* ...]

Description

The ATTACH statement is used to preserve the values of variables used in subprogram(s) between calls to the subprogram(s). When the ATTACH statement is executed, memory space is allocated for the variables and the values are initialized. The variables are not initialized when the subprogram is called and are not destroyed when the subprogram terminates.

An ATTACH statement may appear in the main program or in any subprogram. A subprogram can ATTACH itself. The message `Program not found` is displayed if a specified *sub-name* cannot be found. If a specified *sub-name* is an assembly language subprogram, the message `Bad program type` is displayed.

Attaching a repeatedly used subprogram reduces execution time. However, while the subprogram remains attached, the memory space for the variables remains allocated. ATTACH should be used only when sufficient memory space is available. (Refer to FRE for more information.)

The RELEASE statement is used to release an attached subprogram.

Cross Reference

FRE, RELEASE

(continued)

ATTACH

(continued)

Example

The following program illustrates how to attach a subprogram.

```
100 FOR J=1 TO 5
110 CALL X
120 NEXT J
```

Prints 0 0 0 0 0 because the variable values in subprogram X are initialized each time it is called.

```
130 ATTACH X:PRINT
```

Attaches subprogram X and clears the display.

```
140 FOR J=1 TO 5
150 CALL X
160 NEXT J
```

Prints 0 1 2 3 4 because the variable values are not initialized when X is called and are not destroyed when X is terminated.

```
170 SUB X
180 PRINT J;:PAUSE 2
190 J=J+1
200 SUBEND
```


BREAK

Format

BREAK [*line-number-list*]

Description

The BREAK statement is used to suspend program execution at specific points, called breakpoints, in a program. Breakpoints can be specified in two ways. If *line-number-list* is not given with the BREAK statement, a breakpoint occurs when the BREAK statement is executed. If *line-number-list* is given with the BREAK statement, breakpoint(s) are set immediately before the line(s) listed in *line-number-list*. The **[BREAK]** key also causes the program to stop as if a BREAK statement had been executed.

When a breakpoint occurs, the message BREAK is displayed.

A breakpoint set immediately before a program line remains in the program until the UNBREAK statement is used to remove it or until the line is edited or deleted.

BREAK is useful in debugging a program. When program execution halts at a breakpoint, variables can be printed and calculations can be performed to determine why a program is not executing correctly. The CONTINUE command can be used to resume program execution.

Cross Reference

CONTINUE, ON BREAK, UNBREAK

Examples

```
150 BREAK
```

Causes a breakpoint when the BREAK statement is executed.

```
100 BREAK 120,130
```

Causes breakpoints before execution of lines 120 and 130.

```
BREAK 10,400,130
```

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

CALL

Format

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

Description

The CALL statement transfers control to a subprogram. The first subprogram found with the given *subprogram-name* is executed. After the subprogram is executed, program control returns to the first statement following the CALL statement. The valid types of subprograms are listed below in the order in which the search for the subprogram is performed.

1. Built-in subprograms
2. Assembly language subprograms which are loaded with CALL LOAD
3. BASIC subprograms defined using SUB
4. Subprograms located in *Solid State Software* cartridges

Argument-list is used to pass data to the subprogram. The number and types of arguments in *argument-list* must match the parameters in the *parameter-list* of the subprogram or an error occurs.

Each built-in subprogram is discussed under its own entry in this manual. Assembly language subprograms are discussed in the Editor/Assembler manual. BASIC subprograms are discussed in chapter 4 and in this chapter under SUB.

Cross Reference

SUB

Examples

`CALL CLEANUP`
Deletes unused variable names from the system.

`100 CALL SETLANG(1)`
Changes the language setting to German. This subprogram requires a language number parameter.

SUBPROGRAM

CHAR

Format

CALL CHAR(*character-code*, *pattern-identifier*)

Description

The CHAR subprogram defines special display characters. The characters are defined in a 5-by-8 grid by specifying which dots are "on" and which are "off." Up to seven special characters can be defined at one time. The characters can be displayed by using CHR$ in a DISPLAY or PRINT statement. If a special character is in the display when the pattern definition is changed, the displayed character changes immediately.

Note: Characters defined with the CHAR subprogram are not retained when the computer is turned off.

Character-code specifies which special display character is to be defined. *Character-code* must be a value from 0 through 6.

Pattern-identifier is a string expression whose value defines the pattern for one or more special display characters.

- The first 16 characters of *pattern-identifier* define the specified *character-code*. If *pattern-identifier* is less than 16 characters, the remaining characters are considered to be zeros.
- If *pattern-identifier* is greater than 16 characters, the extra characters define the next sequential *character-code*, until all the *pattern-identifier* characters have been assigned to a *character-code*.
- If *pattern-identifier* has enough characters to define past *character-code* 6, the extra characters are ignored.
- If *pattern-identifier* is a null string, an error occurs.

Each pair of characters in *pattern-identifier* describes the pattern in one row of the grid. The left character is 1 or 0, indicating that the left block is "on" or "off," respectively. The right character is a hexadecimal digit (0 through F) whose binary equivalent is used to determine which dots are on and off, as described above. The rows are described from top to bottom. Note that there is a slight break in the display between the top seven rows and the eighth row.

(continued)

(continued)

The following table shows all possible on/off conditions for each row, and the binary and hexadecimal codes for each condition.

Dot Pattern	Binary Code (0 = Off: 1 = On)	Hexadecimal Code
	00000	00
*	00001	01
*	00010	02
* *	00011	03
*	00100	04
* *	00101	05
* *	00110	06
* * *	00111	07
*	01000	08
* *	01001	09
* *	01010	0A
* * *	01011	0B
* *	01100	0C
* * *	01101	0D
* * *	01110	0E
* * * *	01111	0F
*	10000	10
* *	10001	11
* *	10010	12
* * *	10011	13
* *	10100	14
* * *	10101	15
* * *	10110	16
* * * *	10111	17
* *	11000	18
* * *	11001	19
* * *	11010	1A
* * * *	11011	1B
* * *	11100	1C
* * * *	11101	1D
* * * *	11110	1E
* * * * *	11111	1F

(continued)

(continued)

Cross Reference

CHR$

Examples

To define the dot pattern pictured below, type the following line.

CALL CHAR(0,"04150E04040E1504")

	Left Block	Right Blocks			Hex Codes
ROW 1			*		04
ROW 2	*		*	*	15
ROW 3		*	*	*	0E
ROW 4			*		04
ROW 5			*		04
ROW 6		*	*	*	0E
ROW 7	*		*	*	15
ROW 8			*		04

To display the special character, enter **PRINT CHR$(0)**. Note that the underline cursor also appears in the display in this case.

To define the dot pattern pictured below, type the following line.

CALL CHAR(4,"0A110B12")

	Left Block	Right Blocks			Hex Codes
ROW 1		*		*	0A
ROW 2	*			*	11
ROW 3		*		* *	0B
ROW 4	*			*	12
ROW 5					00
ROW 6					00
ROW 7					00
ROW 8					00

Since rows 5 through 8 are not specified, they are assumed to be zeros.

To display the special character, enter **PRINT CHR$(4)**. Note that the underline cursor also appears in the display in this case.

CHR$

Format

CHR$(*numeric-expression*)

Description

The CHR$ function returns the character corresponding to the ASCII character code specified by *numeric-expression*. The CHR$ function is the inverse of the ASC function. A list of the ASCII character codes for each character in the standard character set is given in appendix D.

CHR$ is used in PRINT and DISPLAY statements to display special characters defined with the CHAR subprogram or the extended character set not available on the keyboard (see appendix D). With peripherals, CHR$ can be used for control operations such as advancing a printer to a new page.

Cross Reference

ASC, CHAR

Examples

```
840 PRINT CHR$(72):PAUSE
```

Prints H.

```
900 X$=CHR$(33)
```

Sets X$ equal to !.

SUBPROGRAM CLEANUP

Format

CALL CLEANUP

Description

The CLEANUP subprogram deletes unused variable names from the system. When CALL CLEANUP is executed, all variable names which are not used in the program currently in memory are removed and all open files are closed. If CALL CLEANUP is executed when a program is stopped at a breakpoint, the CONTINUE command cannot be used to resume program execution.

CALL CLEANUP cannot be used in a program.

CLOSE

Format

CLOSE *#file-number* [, DELETE]

Description

The CLOSE statement terminates the association between a file and its current *file-number*. The file or device cannot be accessed by the program unless it is reopened. After a file is closed, *file-number* can be assigned to another file or device. If an attempt is made to CLOSE a file that is not open, an error occurs.

Any of the following actions close all open files.

- Editing the program or subprogram.
- Entering a NEW, RENUMBER, RUN, OLD, SAVE, or VERIFY command.
- Listing the program to a peripheral device.
- Calling the ADDMEM or CLEANUP subprogram.
- Turning the system off or pressing the reset key.

Normal program termination also closes all open files.

Some peripheral devices allow a file to be deleted at the time it is closed by adding DELETE to the statement. The manual for each peripheral device describes the use of DELETE.

Cross Reference

OPEN

Example

```
790 CLOSE #6
```

Closes file #6.

CONTINUE

Format

CONTINUE [*line-number*]

Description

The CONTINUE (or CON) command is used to resume execution after a breakpoint occurs. A program or subprogram may be continued at the line specified by *line-number*. *Line-number* must refer to a line number in the main program if the main program is stopped. If a subprogram is stopped, *line-number* must refer to a line number in that subprogram. Using an improper *line-number* produces unpredictable results.

The following actions do not allow a CONTINUE to resume execution after a breakpoint:

- Editing the program or subprogram.
- Entering a NEW, OLD, RENUMBER, RUN, SAVE, or VERIFY command.
- Listing the program to a peripheral device.
- Calling the ADDMEM or CLEANUP subprogram.
- Turning the system off or pressing the reset key.

Cross Reference

BREAK

COS

Format

COS(*numeric-expression*)

Description

The COS function calculates the trigonometric cosine of *numeric-expression*. The result is calculated according to the angle units (RAD, DEG, or GRAD) selected prior to using this function. See appendix E for a description of the limits of *numeric-expression*.

Examples

```
140 GRAD
```
Selects GRAD angle setting.

```
150 PRINT COS(30):PAUSE
```
Prints .8910065242.

```
240 RAD
```
Selects RAD angle setting.

```
300 T=COS(PI)
```
Sets T equal to −1.

DATA

Format

DATA *data-list*

Description

The DATA statement is used with the READ statement to assign values to variables. When a READ statement is executed, the values in *data-list* are assigned to the variables specified in the *variable-list* of the READ statement. *Data-list* consists of numeric or string constants, separated by commas. Leading and trailing spaces are ignored. A string constant that contains commas or leading or trailing spaces must be enclosed in quotes. A quotation mark within a quoted string is represented by two quotation marks. A null string is represented by two adjacent commas.

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

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

Cross Reference

READ, RESTORE

(continued)

DATA

(continued)

Example

The program below reads and prints several numeric and string constants.

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

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

```
140 DATA 2,4,6,7,8
150 DATA 1,2,3,4,5
160 DATA """THIS HAS QUOTES"""
170 DATA "NO QUOTES HERE"
180 DATA NO QUOTES HERE EITHER
190 FOR A=1 TO 7
200 READ B$
210 PRINT B$:PAUSE 1.1
220 NEXT A
```

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

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


SUBPROGRAM

DEBUG

Format

CALL DEBUG

Description

The DEBUG subprogram is used to test assembly language subprograms. CALL DEBUG allows access to the assembly language debugger, which is briefly described in appendix I. Refer to the Editor/Assembler manual for more information.

DEG

Format

DEG

Description

The DEG statement sets the units for angle calculations to degrees. After the DEG angle setting is selected, all entered and calculated angles are measured in degrees. This setting is changed to RAD when NEW ALL is entered or the system is initialized.

Cross Reference

GRAD, RAD

DELETE

Format

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

Description

The DELETE (or DEL) statement is used to remove lines from a program in memory or to remove a file from external storage. *Line-group* specifies program lines to be deleted and may consist of the following.

Line-group	Effect
a single line number	Deletes that line.
line number –	Deletes that line and all following lines.
– line number	Deletes that line and all preceding lines.
line number – line number	Deletes that inclusive range of lines.

DELETE *line-group* cannot be used in a program line.

If *line-group* specifies a single line number that does not exist, the message Line not found is displayed. However, any remaining *line-groups* are deleted when the **[ENTER]** or **[CLR]** key is pressed. If the initial line of a range does not exist, the next higher numbered line is used as the initial line. If the final line does not exist, the next lower numbered line is used as the final line.

Device.filename is used to delete a file from an external storage device. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. *Device.filename* may be a string expression. Refer to the peripheral manuals for the device number for each peripheral device and for specific information about the form of *filename*.

You may also delete data files on some peripheral devices by using DELETE in the CLOSE statement. Refer to the appropriate peripheral manual for more information.

(continued)

DELETE

(continued)

Cross Reference

CLOSE

Examples

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

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

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

`DELETE "1.file"`
Deletes "file" from device 1.

DIM

Format

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

Description

The DIM statement specifies the characteristics of an array and reserves the necessary memory space for it. *Array-name* is a string or numeric variable name. The number of values in parentheses following *array-name* determines the number of dimensions in the array. Arrays with up to three dimensions are allowed. The values in parentheses represent the maximum values of the subscripts in each dimension of the array.

The lowest value of a subscript is zero. Therefore, the number of elements in each dimension is one more than the maximum subscript. For example, an array defined by DIM A(6) is a one dimensional array with seven elements, A(0) through A(6). If an array is not defined in a DIM statement, the maximum value of each subscript is 10.

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

An array can be dimensioned only once. A DIM statement must appear in the program at a lower numbered line than any other reference to its array. Remarks (REM) and tail remarks (!) are the only statements which may appear after a DIM statement on a multiple statement line. A DIM statement cannot appear in an IF THEN ELSE statement.

Examples

`120 DIM X$(30)`
Reserves space in the computer's memory for 31 elements of the array called X$. Each element is initialized to the null string.

`430 DIM D(100),B(10,9)`
Reserves space in the computer's memory for 101 elements of the array called D and 110 (11 times 10) elements of the array called B. Each element of each array is initialized to zero.

DISPLAY

Format

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

Description

The DISPLAY statement formats and displays the value(s) included in *print-list*. The options available with DISPLAY can be used to display data starting at any column position, sound an audible tone (beep), erase all or part of the display, limit the total number of characters displayed, and specify the format of the display.

AT(*column*) positions the cursor and the beginning of the display field at the specified column position from 1 through 80.

The evaluation of the TAB function and comma separators is relative to the specified starting position. However, if SIZE is not specified and the evaluation of *print-list* continues on a new line, the new line begins in column 1, not in the column specified by AT.

When AT is omitted, output starts at the current cursor position as left by previous input/output statements. If the current position is greater than 80, the cursor is reset to column 1. When AT is omitted, the TAB function and comma separators are always relative to column 1.

BEEP sounds a short tone for each BEEP in the statement.

ERASE ALL clears the entire 80-column line. If AT is omitted, the cursor position is set to column 1.

SIZE(*numeric-expression*) limits the total number of characters to the absolute value of *numeric-expression*. If *numeric-expression* is larger than the number of remaining positions, the display field extends from the current cursor position to the end of the 80-column line. The length of the display field, defined by SIZE becomes the new record length for purposes of

(continued)

DISPLAY

(continued)

evaluating the TAB function and comma separators in *print-list*. The specified field is always cleared prior to displaying data. Termination of the DISPLAY statement leaves the cursor in the first position following the display field. If SIZE is omitted and there is no trailing separator after *print-list*, termination of the DISPLAY statement clears the display from the last item displayed to the end of the 80-column line.

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

Print-list consists of numeric and string expressions, separated by commas or semicolons. For more details, see PRINT.

Cross Reference

IMAGE, PAUSE, PRINT, TAB, USING

Examples

```
120 DISPLAY AT(7),Y:PAUSE
```

Displays the value of Y starting at column 7 and clears everything following the number. The value actually appears in column 8 since the sign precedes the number.

```
150 DISPLAY N:PAUSE
```

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

```
190 DISPLAY ERASE ALL,B:PAUSE
```

Clears the entire display before displaying the value of B.

```
370 DISPLAY AT(C) SIZE(19) BEEP,X:PAUSE
```

Clears 19 characters starting at position C, beeps, and displays the value of X starting at position C.

END

Format

END

Description

The END statement terminates a program and may be used interchangeably with the STOP statement. Although the END statement may appear anywhere, it is usually placed as the last line in a program. The END statement is not required. A program automatically stops when the highest numbered line is executed.

The END statement closes all open files.

Cross Reference

STOP

EOF

Format

EOF(*file-number*)

Description

The EOF function is used to test whether there is another record to be read from a file. The value of *file-number* indicates the file to be tested and must correspond to the number of an open file. EOF returns a value which indicates the current position in the file as follows.

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

The logical end-of-file occurs when all records on the file have been input.

When using pending INPUT (see chapter 4), EOF does not Indicate whether pending input data remains in memory.

Cross Reference

INPUT (with files)

Examples

```
140 PRINT EOF(3):PAUSE
```

Prints -1 if file #3 has reach the end-of-file and 0 if it has not reached the end-of-file.

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

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

ERR SUBPROGRAM

Format

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

Description

The ERR subprogram returns the error code, error type and, optionally, the file number and line number of the last uncleared error. When an error occurs, a subroutine can be called (see ON ERROR) that contains CALL ERR. The error is cleared when this error-processing subroutine terminates with a RETURN.

Error-codes range from 0 through 127. The meaning of each error code is listed in appendix K.

Error-type is always 0 unless *error-code* is 0, which is an input/output (I/O) error. For an I/O error, *error-type* is an I/O error code specified by each I/O device. The range for I/O error codes is 1 through 255.

File-number is 0 unless the error is an I/O error. For an I/O error, *file-number* is the file number used in the I/O statement that caused the error.

Line-number is the number of the line being executed when the error occurred. It is not always the line that is the source of the problem since an error may occur because of values generated or actions taken elsewhere in a program.

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

Cross Reference

ON ERROR, RETURN (with ON ERROR)

Examples

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

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

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

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

SUBPROGRAM EXEC

Format

CALL EXEC(*execution-address* [, *argument-list*])

Description

The EXEC subprogram is used to execute assembly language subprograms located at specific memory addresses. Normally, the POKE statement has been used to store these subprograms in memory that has been reserved by a CALL GETMEM statement. *Execution-address* is the memory address at which subprogram execution is to begin and must be a numeric expression from 0 through 65535.

Argument-list is used to pass values to and from the subprogram being executed.

Details on writing assembly language subprograms are provided in the Editor/Assembler manual.

Cross Reference

GETMEM, PEEK, POKE, RELMEM

EXP

Format

EXP(*numeric-expression*)

Description

The EXP function returns the result of e^x, where x is *numeric-expression*. The value of e is 2.71828182846.

Examples

```
150 Y=EXP(7)
```

Sets Y equal to the value of e raised to the seventh power, which is 1096.633158429.

```
390 L=EXP(4.394960467)
```

Sets L equal to the value of e raised to the 4.394960467 power, which is 81.04142688867.

FOR TO STEP

Format

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

Description

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

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

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

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

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

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

(continued)

FOR TO STEP

(continued)

Cross Reference

NEXT

Examples

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

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

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

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

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

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

FORMAT

Format

FORMAT *device*

Description

The FORMAT statement initializes the current medium on an external storage device. Formatting a storage medium destroys all previously stored data.

Device is the number associated with each physical device and can be from 1 through 255. Refer to the peripheral manuals to obtain the device code for each peripheral device.

Example

```
140 FORMAT 1
```

Initializes the tape currently in the *Wafertape* drive. All data previously stored on the tape is destroyed.

FRE

Format

FRE(*numeric-expression*)

Description

The FRE function returns information about the current use of memory in the computer. Memory space is divided into four types as follows.

- Space reserved for operation of the system.
- Space occupied by the current program in memory.
- Space temporarily reserved by a running program.
- Space currently available (free space).

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

Value	Meaning
0	Total memory space *not* reserved for system operation.
1	Total space occupied by the program currently in memory. The value returned includes 11 bytes for program overhead.
2	Total amount of free space and temporarily reserved space.
3	Size of the largest block of free memory space.
4	Total amount of free memory space.
5	Number of individual blocks of free memory space.

Example

```
300 A=FRE(3)
```

Sets A equal to the number of bytes available in the largest contiguous block of free memory. This statement is useful for determining how much memory can be reserved by the GETMEM subprogram.

SUBPROGRAM

GETLANG

Format

CALL GETLANG(*numeric-variable*)

Description

The GETLANG subprogram places the code of the international language being used to display system messages and errors into *numeric-variable*. The language identification code is set using the SETLANG subprogram. The language is set to English when the system is initialized.

The following are the assigned language codes.

0 = English
1 = German
2 = French
3 = Italian
4 = Dutch
5 = Swedish
6 = Spanish

Cross Reference

SETLANG

Example

```
120 CALL GETLANG(A)
```

Places the code of the current language setting into A.

GETMEM SUBPROGRAM

Format

CALL GETMEM(*numeric-expression*, *numeric-variable*)

Description

The GETMEM subprogram is used to reserve memory space for storing data and assembly language programs. *Numeric-expression* specifies the number of bytes to reserve and must be a value from 1 through 32765. The error message `Memory full` is displayed if the number of bytes specified is not available.

The lowest address of the reserved memory space is stored in *numeric-variable*. This value must be retained if RELMEM is to be used to release the memory for other uses. The highest address of the reserved memory space can be calculated as follows.

highest memory address = *numeric-variable* + *numeric-expression* – 1

When space has been reserved, CALL POKE and CALL PEEK can be used to access the memory directly. Data may be placed in the reserved area one byte at a time with the CALL POKE subprogram and read with the CALL PEEK subprogram. If an assembly language subprogram is loaded into reserved memory using CALL POKE, CALL EXEC may be used to execute it. The assembly language program must not use memory space outside of the reserved area.

In addition to the requested amount of memory, GETMEM requires four bytes of memory for its own operation. Thus, if the FRE function is used to obtain the size of the largest available block of memory, that value must be reduced by four to obtain the largest block which can be allocated by GETMEM.

The largest block of memory allocated by GETMEM should be significantly less than the largest block available. Sufficient memory space must remain available for statements that require additional temporary memory.

(continued)

SUBPROGRAM GETMEM

(continued)

Cross Reference

EXEC, FRE, PEEK, POKE, RELMEM

Example

```
140 CALL GETMEM(100, ADDRESS)
```

Reserves a block of 100 bytes of memory and places the lowest address of the reserved memory area into ADDRESS.

GOSUB

Format

GOSUB *line-number*

Description

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

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

Cross Reference

ON GOSUB, RETURN

Example

```
100 GOSUB 200
```

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

GOTO

Format

GOTO *line-number*

Description

The GOTO statement transfers control unconditionally to another line within a program. When a GOTO statement is executed, control is passed to the first statement on the line specified by *line-number*.

The GOTO statement cannot be used to transfer control into or out of a subprogram.

Example

```
100 GOTO 300
```

Transfers control to line 300.

GRAD

Format

GRAD

Description

The GRAD statement sets the units for angle calculations to grads. After the GRAD angle setting is selected, all entered and calculated angles are measured in grads. This setting is changed to RAD when NEW ALL is entered or the system is initialized.

Cross Reference

DEG, RAD

IF THEN ELSE

Format

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

Description

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

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

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

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

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

Examples

```
100 IF Y<5 THEN 150
```

If the value of Y is less than 5, statement 150 is executed. If Y is greater than or equal to 5, the next statement is executed.

(continued)

IF THEN ELSE

(continued)

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

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

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

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

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

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

```
290 IF A$="Y" THEN COUNT=COUNT+1:DISPLAY AT(4), "HERE WE
    GO AGAIN!":PAUSE 1.5:GOTO 400
```

If A$ is equal to "Y", COUNT is incremented by 1, a message is displayed, and control is transferred to line 400. If A$ is not equal to "Y", control passes to the next line.

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

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

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

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

IMAGE

Format

IMAGE *string-constant*

Description

The IMAGE statement is used to define an output format. The format is used by placing the line number of the IMAGE statement in the USING option of DISPLAY or PRINT (see USING in this chapter). *String-constant* may be enclosed in quotation marks. If *string-constant* is not enclosed in quotation marks, leading and trailing blanks are ignored.

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

A format definition is divided into format fields and literal fields. When a PRINT or DISPLAY statement uses a format definition, the format fields are replaced by the values of the print items and the literal fields are printed as they appear in the format definition. An explanation of a format definition is given below.

Format Definition

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

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

(continued)

(continued)

Integer Field

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

Decimal Field

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

Exponential Field

- Up to 14 significant digits may be specified.
- An exponential field consists of a decimal or integer field, which defines the mantissa, followed by 4 or 5 exponentiation symbols which define the exponent. When fewer than 4 are used, they are treated as literal characters. When more than 5 are used, the first 5 are used to define the exponential field, and the remainder are considered to be literal characters.
- The number is rounded according to the mantissa definition.

(continued)

(continued)

- When the mantissa definition specifies positions to the left of the decimal point, one of these positions is always used for the sign, which is a minus sign if negative and a space if positive.

String Field

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

Literal Field

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

Cross Reference

DISPLAY, PRINT, USING

(continued)

IMAGE

(continued)

Examples

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

```
100 FOR COUNT=1 TO 6
110 READ A,B
120 PRINT USING 150;A,B:PAUSE
130 NEXT COUNT
140 DATA -99,-9.99,-7,-3.459,0,0,14.8,12.75,795,852,
    -984,64.7
150 IMAGE THE ANSWERS ARE ### AND ##.##
```

The following show the results with the given values.

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

A program similar to the one above allows the use of characters with IMAGE DEAR #####,. The following show the results with certain values.

Value	Appearance
JOHN	DEAR JOHN ,
NANCY	DEAR NANCY,
KENNETH	DEAR *****,

(continued)

IMAGE

(continued)

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

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

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

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

Prints the values using the IMAGE statements.

```
180 NEXT A
190 PRINT USING "$####.##", TOTAL:PAUSE
```

Uses the format directly in the PRINT statement.

INDIC SUBPROGRAM

Format

CALL INDIC(*indicator-number* [, *indicator-state*])

Description

The INDIC subprogram turns the display indicators on or off. *Indicator-number* identifies a specific indicator and must be a numeric expression which rounds to an integer value from 0 through 17.

Indicator-state is used to turn the indicator on or off. A non-zero value turns the indicator on and a zero value turns it off. If *indicator-state* is omitted, the indicator is turned on.

Indicator-numbers 1 through 6 are available for definition in a program. They are turned off when a new program is run or the computer is reset.

The other indicators are used by the system. Changing the status of a system indicator can cause erroneous results.

The numbers assigned to the display indicators are listed below.

Value	Indicator
0	ERROR
1-6	User indicators
7	LOW
8	◀
9	SHIFT
10	CTL
11	FN
12	DEG
13	RAD
14	GRAD
15	I/O
16	UCL
17	▶

WITH KEYBOARD INPUT

Format

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

Description

The INPUT statement is used to enter data from the keyboard. When INPUT is executed, program execution is suspended until data is entered.

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

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

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

When entering numeric variables, a numeric expression can be entered instead of a numeric constant. The expression is evaluated and the result is assigned to the variable. When entering string variables, leading and trailing spaces are ignored. Thus, if a string value includes commas, leading spaces, or trailing spaces, it must be enclosed in quotes. A quotation mark within a quoted string is represented by two quotation marks.

(continued)

INPUT WITH KEYBOARD

(continued)

If **[SHIFT] [ENTER]** is pressed during data entry, the input is ignored and the value of the variable remains unchanged. Execution proceeds to the next prompt or variable or to the next statement if the INPUT statement is completed.

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

When data is entered, the following validations are made.

- If more than one value at a time is entered, the message `Illegal syntax` is displayed and the data must be reentered one at a time.
- If a string constant is entered for a numeric variable, the message `String-number mismatch` is displayed and a numeric value must be entered.
- If a number whose absolute value is greater than 9.9999999999999E + 127 is entered, the message `Overflow` is displayed and the value must be reentered.
- If a number whose absolute value is less than 1E – 128 is entered, the value is replaced with 0 and no message is displayed.

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

(continued)

WITH KEYBOARD INPUT

(continued)

Cross Reference

ACCEPT, INPUT (with files), LINPUT

Examples

`100 INPUT X`

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

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

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

INPUT WITH FILES

Format

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

Description

The INPUT statement is used to read data from files that have been opened in INPUT or UPDATE mode. Each variable in *variable-list* is assigned a value from the file.

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

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

The computer interprets data differently when reading DISPLAY and INTERNAL type data. See "Using External Devices" in chapter 4.

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

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

(continued)

(continued)

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

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

REC *numeric-expression* is used when *file-number* refers to a relative record file. *Numeric-expression* specifies the record to be read from the file. The first record of a file is record zero. See "Using External Devices" in chapter 4 and refer to individual peripheral manuals for information about relative record files and the use of the REC clause.

Cross Reference

CLOSE, INPUT, OPEN, PRINT, RESTORE

(continued)

INPUT WITH FILES

(continued)

Examples

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

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

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

The following program formats the tape in the *Wafertape* peripheral (thereby destroying any data that was previously on the tape), opens it in update mode, and prints five values to the file MYFILE on the tape. The values are then reread and displayed.

```
100 FORMAT 1
110 OPEN #1,"1.MYFILE",INTERNAL, UPDATE
120 FOR A=1 TO 5
130 READ DATAOUT
140 PRINT #1,DATAOUT
```
Lines 120 through 140 read five records from the DATA statement and write them to file #1.

```
150 PRINT DATAOUT;"IS WRITTEN TO FILE #1.":PAUSE 1.5
160 NEXT A
170 RESTORE #1
180 FOR B=1 TO 5
190 INPUT #1,DATAIN
200 PRINT DATAIN;"IS RECORD #";B:PAUSE 1.5
210 NEXT B
```
Lines 180 through 210 read the five records that were written on file #1 and then display their values.

```
220 CLOSE #1, DELETE
```
Deletes the file.

```
230 DATA 15,30,72,36,94
```


INT

Format

INT(*numeric-expression*)

Description

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

Examples

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

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

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

INTRND

Format

INTRND(*numeric-expression*)

Description

The INTRND function returns an integer random number between 1 and the rounded value of *numeric-expression*. The message `Bad argument` is displayed if *numeric-expression* rounds to a value less than one.

This function is equivalent to the expression INT(RND*INT(X+.5))+1.

Examples

```
170 A=INTRND(5*EXP(2))
```

Sets A equal to a random integer value between 1 and 37.

```
330 PRINT INTRND(53):PAUSE
```

Prints a random integer value between 1 and 53.

SUBPROGRAM IO

Format

CALL IO { (*device, command* [, *status-variable*]) | (*string-variable* [, *status-variable*]) }

Description

The IO subprogram performs special control operations which are not available in CC-40 BASIC, but may be supported by some peripherals. Proper use of this subprogram requires knowledge of input/output (I/O) data structures and specific peripheral capabilities. Refer to the peripheral manuals for examples on the use of the IO subprogram and the Editor/Assembler manual for more information.

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

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

String-variable contains from 2 through 12 characters which represent the data required for the I/O operation. The data passed to the IO subprogram are interpreted as binary values. The *string-variable* is always returned with 12 characters. The data length and status may be modified. The format of the string is shown below.

Field Name	Field Length	Description
device	1	peripheral device code
command	1	operation command code
file number	1	file number as assigned in BASIC
record number	2	record number within a file
buffer length	2	size of the buffer for data received from the peripheral
data length	2	number of characters to be sent to the peripheral
status	1	status code returned by the device
buffer pointer	2	highest address of the buffer

(continued)

IO SUBPROGRAM

(continued)

Specific requirements for this data are given in the peripherals manuals and the Editor/Assembler manual.

Status-variable is a numeric variable in which information regarding the result of the operation is stored. If no I/O error occurred, *status-variable* is zero. If an I/O error occurred, *status-variable* contains the corresponding error code. The inclusion of a *status-variable* affects the computer's response to the occurrence of an I/O error. If an I/O error occurs when *status-variable* is given, no error message is displayed and the error cannot be handled by ON ERROR. If an error occurs when *status-variable* is omitted, the message is displayed or the error can be handled by ON ERROR.

Cross Reference

ON ERROR

Example

```
140 CALL IO(1,1)
```

Closes device 1. (A command code of 1 is a CLOSE operation.)

SUBPROGRAM KEY

Format

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

Description

The KEY subprogram assigns the ASCII code of a key pressed from the keyboard to *return-variable*. If no key is pressed, *return-variable* is set equal to 255. See appendix D for a list of the ASCII codes.

Status-variable is used to store a value which represents the status of the key pressed. A value of 1 means a new key was pressed since the last CALL KEY was executed. A value of −1 means the same key was pressed as was returned in the previous CALL KEY. A value of 0 means no key was pressed.

Example

```
340 CALL KEY(K,S)
350 IF S=0 THEN 340
360 PRINT K;CHR$(K)
370 PAUSE
```

Returns in K the ASCII code of any key pressed and in S a value indicating the status of the key pressed.

KEY$

Format

KEY$

Description

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

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

Example

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

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


LEN

Format

LEN(*string-expression*)

Description

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

Examples

```
170 PRINT LEN("ABCDE"):PAUSE
```

Prints 5.

```
230 X=LEN("THIS IS A SENTENCE.")
```

Sets X equal to 19.

```
910 DISPLAY LEN(""):PAUSE
```

Displays 0.

LET

Format

[LET] { *numeric-variable* [, *numeric-variable* ...] = *numeric-expression*
string-variable [, *string-variable* ...] = *string-expression* }

Description

The LET statement assigns the value of an expression to the specified variable(s). The computer evaluates the expression on the right and places the result into the variable(s) on the left. If more than one variable is specified, they must be separated with commas. The LET is optional, and is omitted in the examples in this manual. All subscripts on the left are evaluated before any assignments are made.

Examples

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

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

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

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

LINPUT

Format

LINPUT { [*input-prompt*;] *string-variable*
[#*file-number*, [REC *numeric-expression*,]] *string-variable* }

Description

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

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

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

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

The optional REC clause may be used with devices which support relative record (random access) files. *Numeric-expression* specifies the record to be accessed. Refer to the appropriate peripheral manual for more information concerning relative files.

(continued)

LINPUT

(continued)

Cross Reference

INPUT

Examples

300 LINPUT L$

Causes the computer to display the question-mark prompt and store the entered data in L$.

470 LINPUT "NAME: ";NM$

Causes the computer to display NAME: and store the entered data in NM$.

LIST

Format

LIST { [*line-group*] | ["*device.name*"] | ["*device.name*", *line-group*] }

Description

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

Line-group	Effect
a single line number	Lists that line.
line number –	Lists that line and all following lines.
– line number	Lists that line and all preceding lines.
line number – line number	Lists that inclusive range of lines.

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

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

Examples

LIST 100

Lists line 100 to the display.

LIST 100-200

Lists all lines from 100 through 200 to the display.

LIST "50"

Lists the entire program to peripheral device 50 (presumably a printer).

LIST "50.R=C", -200

Lists all lines up to and including line 200 to peripheral device 50.

LN

Format

LN(*numeric-expression*)

Description

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

Cross Reference

EXP

Examples

```
710 PRINT LN(3.4):PAUSE
```

Prints the natural logarithm of 3.4, which is 1.223775432.

```
850 X=LN(EXP(2.7))
```

Sets X equal to the natural logarithm of e raised to the 2.7 power, which equals 2.7.

```
910 S=LN(SQR(T))
```

Sets S equal to the natural logarithm of the square root of the value of T.

SUBPROGRAM

LOAD

Format

CALL LOAD("*device.filename*")

Description

The LOAD subprogram loads assembly language subprograms from an external storage device into computer memory. These subprograms are run using the CALL EXEC statement.

More than one subprogram may be loaded into memory. When space permits, assembly language subprograms may reside in memory in addition to BASIC programs and subprograms. When loaded in this manner, these subprograms are appended to the memory space reserved for system operation.

Device.filename identifies the device where the assembly language subprogram is stored and the particular file to be loaded. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. An error occurs if the LOAD subprogram determines that the contents of the specified file are not an assembly language subprogram. Refer to the appropriate peripheral manuals for the proper device code and for specific information about the form of *filename*.

Loaded subprograms remain in memory until NEW ALL is entered or the system is initialized.

Cross Reference

EXEC

Examples

```
CALL LOAD("1.MYSUBS")
```

Loads the subprogram in file MYSUBS on device 1 into memory.

```
100 INPUT "ENTER FILE NAME",A$
110 CALL LOAD ("1."&A$)
```

Loads the assembly language subprogram entered by the program user.

LOG

Format

LOG(*numeric-expression*)

Description

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

Examples

```
150 PRINT LOG(3.4):PAUSE
```

Prints the common logarithm of 3.4, which is .531478917.

```
230 S=LOG(SQR(T))
```

Sets S equal to the common logarithm of the square root of the value of T.

NEW

Format

NEW [ALL]

Description

The NEW command prepares the computer for a new program by deleting the program and variables currently in memory. All open files are closed.

The NEW ALL command deletes the current program and variables in memory, clears the user-assigned strings and assembly language subprograms, cancels any expansion of memory implemented by CALL ADDMEM, clears all display indicators, sets the angle mode to RAD, and closes all open files.

NEXT

Format

NEXT [*control-variable*]

Description

The NEXT statement is always paired with a FOR TO STEP statement for construction of a loop. If *control-variable* is given, it must be the same as *control-variable* in the FOR TO STEP statement. If *control-variable* is omitted, NEXT is paired with the most recent, unmatched FOR TO STEP statement. It is good programming practice to include *control-variable*.

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

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

Cross Reference

FOR TO STEP

Example

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

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


NUMBER

Format

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

Description

The NUMBER (or NUM) command generates sequenced line numbers. These line numbers are displayed with a trailing space for convenience when entering program lines. All that needs to be typed in are the statement(s). After **[ENTER]** is pressed, the line is stored in memory and the next line number is displayed.

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

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

Cross Reference

RENUMBER

Examples

`NUM 110`

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

`NUM 105,5`

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

NUMERIC

Format

NUMERIC(*string-expression*)

Description

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

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

Cross Reference

VAL

Example

The following program segment determines if an entry from the keyboard is a valid numeric constant. If it is not, an error message is displayed until data is reentered. If the data is a numeric constant, it is stored in variable A.

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


OLD

Format

OLD "*device.filename*"

Description

The OLD command loads a program from an external device into memory. OLD closes all open files and removes the program currently in memory before loading the program. A BASIC program can be stored on *device.filename* with the SAVE command.

Device.filename identifies the device where the program is stored and the name of the file. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the particular file. Refer to the peripheral manuals for the device code for each peripheral device and for specific information about the form of *filename*.

Note: If *filename* specifies a data file rather than a program file, it may be necessary to press the reset key.

Cross Reference

SAVE

Example

```
OLD "1.MYPROG"
```

Loads the program MYPROG into the computer's memory from peripheral device 1.

ON BREAK

Format

ON BREAK { STOP / NEXT / ERROR }

Description

The ON BREAK statement determines the action taken when a breakpoint occurs. After the ON BREAK statement is executed, breakpoints are handled according to the option selected.

ON BREAK STOP restores the normal function of BREAK, which is to halt program execution and display the standard breakpoint message. This option is set when a program is run.

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

ON BREAK ERROR causes breakpoints to be treated as errors, which allows the ON ERROR statement to be used to process breakpoints. See ON ERROR for more information.

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

Cross Reference

BREAK, ON ERROR

(continued)

ON BREAK

(continued)

Example

The program below illustrates the use of ON BREAK. When the message Break is displayed, press **[CLR]** and enter **CONTINUE**.

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

ON ERROR

Format

ON ERROR {STOP / *line-number*}

Description

The ON ERROR statement determines the action taken when an error occurs during the execution of a program. After the ON ERROR statement is executed, any errors that occur are handled according to the option selected.

ON ERROR STOP restores the normal way of handling errors which is to halt program execution and print a descriptive error message. This option is set when a program is run.

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

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

The main program and subprograms can share the same error-processing subroutine. Subroutines called by GOSUB cannot be shared.

Cross Reference

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

(continued)

ON ERROR

(continued)

Example

The program below illustrates the use of ON ERROR.

```
100 ON ERROR 150
```
Causes any error to pass control to line 150.
```
110 X$="A"
120 X=VAL(X$)
```
Causes an error.
```
130 PRINT X;"SQUARED IS";X*X:PAUSE 2
140 STOP
150 REM ERROR SUBROUTINE
160 ON ERROR 220
```
Causes the *next* error to pass control to line 220.
```
170 CALL ERR(CODE,TYPE,FILE,LINE)
```
Determines the error using CALL ERR.
```
180 IF LINE< >120 THEN RETURN 220
```
Transfers control to line 220 if the error is not in the expected line.
```
190 IF CODE< >29 THEN RETURN 220
```
Transfers control to line 220 if the error is not the one expected.
```
200 X$="5"
```
Changes the value of X$ to an acceptable value.
```
210 RETURN
```
Returns control to the line in which the error occurred.
```
220 REM UNKNOWN ERROR
230 PRINT "ERROR";CODE;" IN LINE";LINE:PAUSE
```
Reports the nature of the unexpected error and the program stops.

ON GOSUB

Format

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

Description

The ON GOSUB statement determines which subroutine to execute by evaluating *numeric-expression*. If the value of *numeric-expression* is 1, the subroutine starting at *line-number1* is executed; if 2, the subroutine starting at *line-number2* is executed, and so forth. Each line number must be the first statement of a subroutine. If *numeric-expression* is 0, negative, or larger than the list of line numbers, the error message `Bad value` is displayed. If *numeric-expression* is a decimal number, it is rounded.

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

Cross Reference

GOSUB, RETURN (with GOSUB)

Examples

```
140 ON X GOSUB 1000,2000,300
```

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3.

```
240 ON P-4 GOSUB 200,250,300,800,170
```

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

ON GOTO

Format

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

Description

The ON GOTO statement determines where to transfer control by evaluating *numeric-expression*. If the value of *numeric-expression* is 1, control is transferred to *line-number1*; if 2, control is transferred to *line-number2*, and so forth. If *numeric-expression* is 0, negative, or greater than the list of line numbers, the error message `Bad value` is displayed. If *numeric-expression* is a decimal number, it is rounded.

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

Cross Reference

GOTO

Examples

```
130 ON X GOTO 1000,2000,300
```

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3. The equivalent statement using an IF THEN ELSE statement is `130 IF X=1 THEN 1000 ELSE IF X=2 THEN 2000 ELSE IF X=3 THEN 300 ELSE PRINT "Bad value": PAUSE:STOP`, which is more than 80 characters.

```
210 ON P-4 GOTO 200,250,300,800,170
```

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

ON WARNING

Format

ON WARNING { PRINT / NEXT / ERROR }

Description

The ON WARNING statement determines the action taken when a warning occurs during the execution of a program. After the ON WARNING statement is executed, any warning is handled according to the ON WARNING option selected.

ON WARNING PRINT restores the normal use of warnings which is to print a descriptive warning message and continue program execution after the **[ENTER]** or **[CLR]** key is pressed. This option is selected when a program is run.

ON WARNING NEXT causes the program to continue execution without printing any message.

ON WARNING ERROR causes the occurrence of a warning to be treated as an error, allowing effective handling of warnings with ON ERROR statements.

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

Cross Reference

ON ERROR

(continued)

ON WARNING

(continued)

Example

The program below illustrates the use of ON WARNING.

```
100 ON WARNING NEXT
```
Sets warning handling to go to the next statement.
```
110 PRINT 110,5/0:PAUSE
```
Prints the result without any message.
```
120 ON WARNING PRINT
```
Sets warning handling to the normal option, which is to print a message and allow execution to continue when a warning occurs.
```
130 PRINT 130,5/0:PAUSE
```
Prints the warning. When **[ENTER]** or **[CLR]** is pressed, prints 130 followed by the value of 5/0.
```
140 ON WARNING ERROR
```
Sets warning handling to treat warnings as errors.
```
150 PRINT 150,5/0:PAUSE
```
Prints the warning message and treats the warning as an error.
```
160 PRINT 160:PAUSE
```
Not executed because execution stops in line 150.

OPEN

Format

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

Description

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

If an OPEN statement references a file that already exists, the attributes in the OPEN statement must be the same as the attributes of the file.

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

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

(continued)

OPEN

(continued)

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

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

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

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

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

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

(continued)

OPEN

(continued)

Cross Reference

CLOSE, INPUT, LINPUT, PRINT, RESTORE (Also see chapter 4.)

Examples

```
100 OPEN #23,"1.X",INTERNAL,UPDATE
```

Opens the file named "X" on peripheral device 1 and enables any input/output statement to access the file by using the number 23. The type of the file is INTERNAL. Since the file is opened in UPDATE mode, data can be both read from and written to the file.

```
150 OPEN #243,A$&".ABC",INTERNAL
```

If A$ equals "1", opens a file on device 1 with a name of ABC. The file type is INTERNAL, UPDATE mode is assumed, and the device specifies the default record length.

PAUSE

Format

PAUSE { [*numeric-expression*] / [ALL] }

Description

The PAUSE statement suspends program execution either for a specified number of seconds or until the **[CLR]** or **[ENTER]** key is pressed. If *numeric-expression* is omitted, the underline cursor is displayed in column one to indicate an indefinite pause is occurring. The cursor control keys can then be used to view the contents of the 80-column line. Execution continues when either **[ENTER]** or **[CLR]** is pressed.

If *numeric-expression* is given, PAUSE suspends program execution for the number of seconds in the absolute value of *numeric-expression*. If *numeric-expression* is positive, the timed pause can be overridden by pressing **[ENTER]** or **[CLR]**. If negative, the timed pause cannot be overridden. The effective resolution is approximately one tenth of a second. If *numeric-expression* is less than .1, the program does not pause. During a timed pause, the cursor is not displayed and the display cannot be scrolled.

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

PAUSE ALL remains in effect when a subprogram is called. If PAUSE ALL is modified in a subprogram, it is again in effect when the subprogram ends.

Cross Reference

DISPLAY, PRINT

(continued)

PAUSE

(continued)

Examples

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

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

The following program changes degrees Fahrenheit to degrees Celsius.

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

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

```
130 GOTO 100
```


SUBPROGRAM

PEEK

Format

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

Description

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

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

Cross Reference

POKE

Example

```
100 CALL PEEK(2096,X1,X2,X3,X4)
```
Returns the values in locations 2096, 2097, 2098, and 2099 in variables X1, X2, X3, and X4, respectively.

PI

Format

PI

Description

The PI function returns the value of π as 3.14159265359.

Example

```
130 VOLUME=4/3*PI*R^3
```

Sets VOLUME equal to four thirds times PI times the radius cubed, which is the volume of a sphere with a radius of R.

SUBPROGRAM

POKE

Format

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

Description

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

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

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

Cross Reference

PEEK

Example

```
200 CALL POKE(ADDR,162,10,17)
```

Places the values 162, 10, and 17 in the locations ADDR, ADDR + 1, and ADDR + 2 respectively.

POS

Format

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

Description

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

Examples

```
110 X=POS("PAN","A",1)
```

Sets X equal to 2 because A is the second letter in PAN.

```
140 Y=POS("APAN","A",2)
```

Sets Y equal to 3 because the A in the third position in APAN is the first occurrence of A in the portion of APAN that was searched.

```
170 Z=POS("PAN","A",3)
```

Sets Z equal to 0 because A was not in the part of PAN that was searched.

```
290 R=POS("PABNAN","AN",1)
```

Sets R equal to 5 because the first occurrence of AN starts with the A in the fifth position in PABNAN.

WITH DISPLAY

PRINT

Format

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

Description

The PRINT statement may be used to format and write data to the display. USING may be used to specify a format for the items in *print-list*. Refer to IMAGE and USING for a description of format definition and its effect upon the PRINT statement. If *print-list* is omitted, the PRINT statement clears the display.

Print-list consists of print items and print separators. Print items are numeric and string expressions that are displayed and TAB functions that control print positioning. Print separators are commas or semicolons that indicate the position of print items in the display.

Print Items

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

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

(continued)

(continued)

Print Separators

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

- The semicolon prints the next item in the *print-list* immediately after the last print item, with no extra spaces between the values.
- The comma prints the next print item at the beginning of the next print field. The print fields are 15 characters long and are located at columns 1, 16, 31, 46, 61, and 76 for an 80-column line. If the current column position is past the start of the last print field, the comma causes the next printed item to be displayed in the next line.

If a print item is longer than the remainder of the current line, it is displayed at the start of the next line. If a numeric print item fits on the current line without its trailing space, it is printed on the current line. If a print item is longer than 80 characters, the first 80 characters are printed on one line and the remaining characters are printed on successive lines, 80 characters at a time.

Pending Prints

If the *print-list* is not followed by a comma or a semicolon, the remainder of the 80-column line is cleared. Therefore, the next input/output statement must begin a new line.

Using a comma or a semicolon after *print-list* creates a pending print which causes the remainder of the line not to be cleared. Instead the computer spaces over to the start of the next field if a comma ended the PRINT statement, or does not space at all if a semicolon ended the statement. The next I/O statement displays or accepts information beginning at the current column position unless the statement changes the position.

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

(continued)

(continued)

Numeric Formats

Numbers are printed in either normal decimal form or scientific notation. Scientific notation is used when more significant digits can be shown.

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

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

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

mantissa E exponent

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

- The mantissa is printed with 7 or fewer digits with one digit always to the left of the decimal.
- Trailing zeros are omitted in the fractional part of the mantissa.
- The exponent is displayed with a plus or minus sign followed by a two or three digit exponent.
- When the exponent is two digits, the mantissa is limited to seven digits. When the exponent is three digits, the mantissa is limited to six digits. When necessary, the mantissa is rounded to the appropriate number of digits.

Cross Reference

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

(continued)

PRINT WITH DISPLAY

(continued)

Examples

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

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

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

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

PRINT WITH FILES

Format

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

Description

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

REC *numeric-expression* may appear only when *file-number* refers to a relative record file. Refer to chapter 4 and the individual peripheral manuals for information about relative record files and the proper use of REC. *Numeric-expression* is evaluated to designate the specific record number of the file to which to write.

USING may be used to specify an exact format for a display-type file. Refer to the IMAGE and USING sections for a description of format definition and its effect upon the PRINT statement. Including USING in a reference to an internal-type data file results in an error.

Print-list consists of print items and print separators. Print items are numeric and string expressions that are displayed and TAB functions that control print positioning. Print separators are commas or semicolons that indicate the position of print items in the display.

Print-list is interpreted in order from left to right. The form of the output depends upon the type (DISPLAY or INTERNAL) of file or device. See OPEN and chapter 4 for a description of *file-type*.

(continued)

(continued)

Display-type Files

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

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

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

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

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

(continued)

(continued)

Internal-type Files

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

- String expressions are evaluted and printed in the record in internal string representation.
- Numeric expressions are evaluated and printed in the record in internal numeric representation.
- The TAB function causes an error when used in printing to an internal-type file.

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

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

If a print item is longer than the remainder of the current record, the current record is printed and the print item is written at the start of the next record. If a print item is longer than the record length, an error occurs.

Pending Prints

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

Using a comma or a semicolon after *print-list* creates a pending print. If the *print-list* ends with a comma or semicolon, the current record is not written. The computer spaces over to the start of the next field if a comma ended the PRINT statement, or does not space at all if a semicolon ended the statement. The next output statement which accesses this file prints data on this same record, beginning at the current column position unless the statement changes the position.

(continued)

PRINT WITH FILES

(continued)

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

Cross Reference

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

Examples

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

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

The program below writes data to a file.

```
100 OPEN #5,"1.MYPROG",INTERNAL,UPDATE
```

Opens file number 5. MYPROG is created if it does not already exist on device number 1.

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

Dimensions an array for 51 values.

```
120 B=0
```

Initializes the summation variable.

```
130 FOR J=1 TO 50
```

Lines 130 through 180 facilitate data input.

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

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

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

Value of summation variable is written to the file.

```
200 CLOSE #5
```


RAD

Format

RAD

Description

The RAD statement sets the units for angle calculations to radians. After the RAD angle setting is selected, all entered and calculated angles are measured in radians. The RAD setting is selected when NEW ALL is entered or the system is initialized.

Cross Reference

DEG, GRAD

RANDOMIZE

Format

RANDOMIZE [*numeric-expression*]

Description

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

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

Example

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

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


READ

Format

READ *variable-list*

Description

The READ statement is used with the DATA statement to assign values to variables. *Variable-list* consists of string and numeric variables, either subscripted or unsubscripted, separated by commas. The value read in the DATA statement must correspond to the type of the variable to which it is assigned in READ. Note that any number is a valid string. When two adjacent commas are encountered in the data list, a null string is read.

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

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

READ can read data only from a DATA statement that is in the same program or subprogram as the READ statement. Each time a subprogram is called, data is read from the first DATA statement whether or not the subprogram has been attached. (See ATTACH in this chapter.)

Cross Reference

ATTACH, DATA, RESTORE

RELEASE

Format

RELEASE *sub-name1* [, *sub-name2* ...]

Description

The RELEASE statement is used to release attached subprograms. (See ATTACH in this chapter). When RELEASE is executed, the allocated memory space for the subprogram variables is released, and thus the values are destroyed.

Releasing a repeatedly used subprogram increases execution time for a program. However, the subprogram variables do not require memory space between calls to the subprogram.

A RELEASE statement may appear in the main program or in any subprogram, including a subprogram that it releases. If a *sub-name* is specified for an active subprogram, the variables are not released until the subprogram terminates. If *sub-name* specifies an assembly language program, an error occurs. If a specified *sub-name* is not attached or does not exist, that *sub-name* parameter is ignored.

Cross Reference

ATTACH

(continued)

RELEASE

(continued)

Example

The following program illustrates the use of the RELEASE statement.

```
100 ATTACH X
110 FOR J=1 TO 5
120 CALL X
130 NEXT J
```

Prints 0 1 2 3 4 because the variable values are not initialized when X is called and are not destroyed when X is terminated.

```
140 RELEASE X:PRINT
```

Releases subprogram X and clears the display.

```
150 FOR J=1 TO 5
160 CALL X
170 NEXT J
```

Prints 0 0 0 0 0 because the variable values in subprogram X are initialized each time it is called.

```
180 SUB X
190 PRINT J;:PAUSE 2
200 J=J+1
210 SUBEND
```

RELMEM

Format

CALL RELMEM(*numeric-expression*)

Description

The RELMEM subprogram releases memory previously reserved by the GETMEM subprogram. The value given in *numeric-expression* must be the same address returned by GETMEM when the memory space was reserved. If the wrong value is specified for *numeric-expression*, the contents of memory, including the program, can be lost.

Cross Reference

GETMEM, PEEK, POKE

Example

The following example acquires some memory with a CALL GETMEM. POKE is used to store an assembly language program in the memory. When the subprogram is no longer needed, the memory is returned to the system with CALL RELMEM.

```
100 CALL GETMEM (50,ADDR)
110 CALL POKE (ADDR,...)
120 CALL EXEC (ADDR)
    .
    .
220 CALL RELMEM(ADDR)
```


REM

Format

REM [*character-string*]

Description

The REM statement allows you to enter explanatory remarks into your program. Remarks may give any type of information, but usually explain a section of a program. *Character-string* may include any displayable character.

Remarks are not executed, but they do take up space in memory. Any character that follows REM, including the statement separator symbol (:) is considered part of the remark. Therefore, if REM is part of a multiple statement line, it must be the last statement on the line.

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

Example

```
150 REM BEGIN SUBROUTINE
```

Identifies a section beginning a subroutine.

```
270 SUBTOTAL=L+B ! Calculate subtotal
```

Identifies statements which perform a specific calculation.

RENUMBER

Format

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

Description

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

REN also changes all references to line numbers so that they refer to the same lines of code as before. If a statement refers to a line number that does not exist, a warning is displayed and the line number is replaced with 32767, which is not a valid line number.

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

Example

```
REN
```

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

RESTORE

Format

RESTORE { [*line-number*] | [#*file-number* [, REC *numeric-expression*]] }

Description

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

RESTORE specifies that the next READ statement executed accesses the first item in the DATA statement specified by *line-number*. *Line-number* must be in the same program or subprogram as the RESTORE statement. If no *line-number* is given, the DATA statement with the lowest numbered line in the current program or subprogram is used. If *line-number* is not a DATA statement, the next DATA statement following it is used.

RESTORE #*file-number* positions that file to the first record. The next input/output statement that refers to *file-number* accesses the first record in the file. Any pending output data is written to the file before the RESTORE statement is executed. Any pending input data is ignored. *File-number* 0 refers to a DATA statement as described above.

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

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

(continued)

RESTORE

(continued)

Cross Reference

DATA, INPUT, LINPUT, PRINT, READ

Examples

150 RESTORE

Sets the next DATA statement to be read to the first DATA statement in the program.

200 RESTORE 130

Sets the next DATA statement to be read to the DATA statement at line 130 or, if line 130 is not a DATA statement, to the next DATA statement after line 130.

230 RESTORE #1

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

RETURN

WITH GOSUB

Format

RETURN

Description

RETURN used with GOSUB transfers control back to the statement following the GOSUB or ON GOSUB statement which was last executed. A subroutine may contain more than one RETURN statement.

Cross Reference

GOSUB, ON GOSUB

RETURN WITH ON ERROR

Format

RETURN {[NEXT] / [*line-number*]}

Description

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

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

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

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

Cross Reference

ON ERROR

(continued)

RETURN WITH ON ERROR

(continued)

Example

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

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

RND

Format

RND

Description

The RND function returns the next pseudo-random number in the current sequence of pseudo-random numbers. The number returned is greater than or equal to zero and less than one. Unless the RANDOMIZE statement is used to create an unpredictable sequence, RND generates the same sequence each time a program is run.

Cross Reference

INTRND, RANDOMIZE

Example

```
100 PRINT 10*RND :PAUSE
```

Prints a random number greater than or equal to 0 and less than 10.

RPT$

Format

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

Description

The RPT$ function returns a string that is *numeric-expression* repetitions of *string-expression*. If RPT$ produces a string longer than 255 characters, the excess characters are discarded and the warning message `String-truncation` is displayed.

Examples

```
100 M$=RPT$("ABCD",4)
```

Sets M$ equal to "ABCDABCDABCDABCD".

```
100 CALL CHAR(0,RPT$("0000FFFF",8))
```

Defines characters 0 through 3 with the string "0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF 0000FFFF0000FFFF0000FFFF".

```
100 PRINT USING RPT$("#",40);X$ :PAUSE
```

Prints the value of X$ using an image that consists of 40 number signs.

RUN

Format

RUN { [*line-number*] / ["*program-name*"] / ["*device.filename*"] }

Description

The RUN statement starts execution of a program. The statement RUN entered with no options starts execution of the program currently in memory beginning with the lowest numbered line.

RUN *line-number* starts execution of the program in memory at the specified *line-number.*

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

RUN "*device.filename*" deletes the program currently in memory, loads the contents of *filename* from *device* into memory, and executes it. A string expression may be used to specify *device.filename.* Note: If *filename* specifies a data file rather than a program file, it may be necessary to press the reset key.

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

- Variables are initialized. Numeric variables are set to zero and string variables are set to null strings.
- Certain errors, such as a FOR statement without a NEXT statement or a line reference out of range, are detected.
- All open files are closed.
- ON BREAK STOP, ON WARNING PRINT, and ON ERROR STOP are selected.
- The angle mode selected is left unchanged.

(continued)

RUN

(continued)

Examples

`RUN`

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

`RUN 200`

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

`RUN "1.PRG3"`

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

`RUN "STAT"`

Executes the program STAT in the *Solid State Software* cartridge.

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

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

SAVE

Format

SAVE "*device.filename*" [, PROTECTED]

Description

The SAVE command allows you to copy the BASIC program in memory to an external device. SAVE removes any variables from the system which are not used in the program. By using the OLD command, you can later recall the program into memory.

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

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

Cross Reference

OLD, VERIFY

Examples

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

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

SEG$

Format

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

Description

The SEG$ function returns a substring of a string. The string returned starts at *position* in *string-expression* and extends for *length* characters. If *position* is beyond the end of *string-expression*, the null string ("") is returned. If *length* extends beyond the end of *string-expression*, only the characters through the end are returned.

Examples

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

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

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

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

SETLANG SUBPROGRAM

Format

CALL SETLANG(*numeric-expression*)

Description

The SETLANG subprogram selects the language in which system messages and errors are displayed. *Numeric-expression* is a number that is the code of a specific language. The following are the assigned language codes.

0 = English
1 = German
2 = French
3 = Italian
4 = Dutch
5 = Swedish
6 = Spanish

If *numeric-expression* is 0 or 1, all system messages and errors are displayed in English or German, respectively. If *numeric-expression* selects any other language that is supported in a *Solid State Software*™ cartridge, prompts and messages in the cartridge are displayed in the chosen language, but all system messages and errors are displayed in English.

The language code is maintained by the *Constant Memory*™ feature. Therefore, the language code setting is not altered by turning the computer on or off, and remains in effect until it is changed or the system is initialized. When the system is initialized, the language code is set to zero (English).

Cross Reference

GETLANG

SGN

Format

SGN(*numeric-expression*)

Description

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

Examples

```
140 IF SGN(A)=1 THEN 300 ELSE 400
```

Transfers control to line 300 if A is positive and to line 400 if A is zero or negative.

```
790 ON SGN(X)+2 GOTO 200,300,400
```

Transfers control to line 200 if X is negative, line 300 if X is zero, and line 400 if X is positive.

SIN

Format

SIN(*numeric-expression*)

Description

The sine function gives the trigonometric sine of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current angle mode in effect (see DEG, GRAD, and RAD). See appendix E for a description of the limits of *numeric-expression*.

Example

```
150 DEG
160 PRINT SIN(3*21.5+4):PAUSE
```

Prints .930417568.

SQR

Format

SQR(*numeric-expression*)

Description

The SQR function returns the positive square root of *numeric-expression*. SQR(X) is equivalent to X^(1/2). *Numeric-expression* must not be a negative number.

Examples

```
150 PRINT SQR(4):PAUSE
```

Prints 2.

```
780 X=SQR(2.57E5)
```

Sets X equal to the square root of 257,000, which is 506.9516742255.

STOP

Format

STOP

Description

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

Cross Reference

END

Example

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

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


STR$

Format

STR$(*numeric-expression*)

Description

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

Cross Reference

LEN, VAL

Examples

```
150 NUM$=STR$(78.6)
```

Sets NUM$ equal to "78.6".

```
220 LL$=STR$(3E15)
```

Sets LL$ equal to "3.E+15".

```
330 J$=STR$(A*4)
```

Sets J$ equal to a string equal to the value obtained when A is multiplied by 4. For instance, if A is equal to −8, J$ is set equal to "−32".

SUB

Format

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

Description

The SUB statement is the first statement in a subprogram and must be the first statement on the line. A subprogram is a group of statements separated from the main program. A subprogram is used to perform the same task in several different places without duplicating the statements in several places.

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

Parameter-list defines the information passed to the subprogram. A parameter may be a simple string variable, a simple numeric variable, or an array. An array is listed as a parameter by writing the array name followed by parentheses. A one-dimensional array is written as A( ), a two-dimensional array as A(,), and a three-dimensional array as A(,,).

Information is passed to the subprogram through the *argument-list* of the CALL statement. The arguments of *argument-list* and the parameters of *parameter-list* need not have the same names. However, the number and the types of arguments in *argument-list* must match the number and types of parameters in *parameter-list* of the SUB statement.

(continued)

SUB

(continued)

Information is passed to a subprogram either by reference or by value. If an argument is passed by reference, the subprogram uses the variables from the calling program. If the corresponding parameter in the subprogram is changed, the argument in the calling program is also changed. A simple variable, an element of an array, or an array listed in *argument-list* is passed by reference. Arrays are always passed by reference.

If an argument is passed by value, only the value of the argument is passed to the subprogram. If the corresponding parameter in the subprogram is changed, it does not alter the value of the argument in the calling program. Any type of expression in *argument-list* is evaluated and passed by value to the subprogram. Simple variables may be passed by value by enclosing them in parentheses.

All variables used in a subprogram other than those in *parameter-list* are local to that subprogram, so the same variable names may be used in the main program and in other subprograms. Changing the values of local variables in a program or subprogram does not affect the values of local variables in any other program or subprogram.

Any local variables in the subprogram are initialized each time the subprogram is called, unless the subprogram has been attached. Attaching a subprogram causes the values of the variables to be retained between calls until the subprogram has been released. See ATTACH and RELEASE.

A subprogram terminates when a SUBEXIT or SUBEND statement is executed. Control is returned to the statement following the CALL statement.

(continued)

SUB

(continued)

Subprograms appear after the main program. A subprogram cannot contain another subprogram. When a SUB statement is encountered in a main program, it terminates as if a STOP statement had been executed. Only remarks and END statements may appear between the SUBEND of one program and the SUB of the next subprogram.

The ON BREAK, ON WARNING, ON ERROR, and PAUSE ALL statements in effect when a CALL is executed remain in effect while the subprogram is executing. If the subprogram changes any of these statements, they are changed back when the subprogram terminates. Subprograms cannot share any subroutines except error-processing subroutines.

Cross Reference

ATTACH, CALL, ON BREAK, ON ERROR, ON WARNING, RELEASE, RETURN, SUBEND, SUBEXIT

(continued)

SUB

(continued)

Examples

100 SUB MENU

Marks the beginning of a subprogram. No parameters are passed or returned.

220 SUB MENU(COUNT,CHOICE)

Marks the beginning of a subprogram. The variables COUNT and CHOICE may be used and/or have their values changed in the subprogram and their corresponding arguments in the calling statement changed.

330 SUB PAYCHECK(DATE,(Q),SSN,PAYRATE,TABLE(,))

Marks the beginning of a subprogram. The variables DATE, SSN, PAYRATE, and the array TABLE with two dimensions may be used and/or have their values changed in the subprogram and their corresponding arguments in the calling statement changed. The variable Q cannot be altered by the subprogram.

SUBEND

Format

SUBEND

Description

The SUBEND statement marks the end of a subprogram. When SUBEND is executed, control is passed to the statement following the statement that called the subprogram. The SUBEND statement must always be the last statement in a subprogram and cannot be in an IF THEN ELSE statement. Only remarks and END statements may appear between a SUBEND statement and the next SUB statement.

Cross Reference

SUB, SUBEXIT

SUBEXIT

Format

SUBEXIT

Description

The SUBEXIT statement terminates execution of a subprogram. When it is executed, control is passed to the statement following the statement that called the subprogram. The SUBEXIT statement may appear as many times as needed in a subprogram.

Cross Reference

SUB, SUBEND

TAB

Format

TAB(*numeric-expression*)

Description

The TAB function is used in a PRINT or DISPLAY statement to select a specific column position for a printed item. If *numeric-expression* is less than or equal to zero, the position is set to one. If *numeric-expression* is greater than the length of a record for the device being used, then *numeric-expression* is repeatedly reduced by the record length until it is less than the record length.

If the current position is less than or equal to the specified position, the TAB function spaces over to the specified position. If the current position is greater than the specified position, the TAB function proceeds to the next record and spaces over to the specified position.

The TAB function is treated as a *print-item* and must be separated from other print items by a print separator. The print separator before TAB is evaluated before the TAB function and the print separator following TAB is evaluated after the TAB function. Normally, semicolons are used before and after TAB.

In a DISPLAY statement, the TAB function is relative to the beginning of the display field. If AT is used, the TAB function is relative to the specified column position. If more than one line of output is displayed, subsequent lines begin in column one. Any TAB functions are then relative to column one.

If SIZE is used, the value specified in SIZE is the absolute limit of the number of characters displayed. This limit is the record length used in evaluating any TAB functions.

(continued)

TAB

(continued)

Cross Reference

DISPLAY, PRINT

Examples

```
100 PRINT TAB(12);35 :PAUSE
```
Prints the number 35 starting at column 13.

```
190 PRINT 356;TAB(18);"NAME" :PAUSE
```
Prints 356 at the beginning of the line and NAME starting at column 18.

```
710 DISPLAY AT(10) SIZE(20),"MGB";TAB(10);"ADDR" :PAUSE
```
Prints MGB starting at column 10 and ADDR starting at column 19.

TAN

Format

TAN(*numeric-expression*)

Description

The TAN (tangent) function gives the trigonometric tangent of *numeric-expression*. The expression is interpreted as radians, degrees, or grads according to the current angle mode in effect (see DEG, GRAD, and RAD). See appendix E for a description of the limits of *numeric-expression*.

Example

```
250 RAD
260 PRINT TAN(20):PAUSE
```

Prints 2.237160944.

UNBREAK

Format

UNBREAK [*line-list*]

Description

The UNBREAK statement removes all breakpoints. If *line-list* is specified, only the breakpoints for those lines listed are removed.

Cross Reference

BREAK

Examples

```
UNBREAK
```

Removes all breakpoints.

```
400 UNBREAK 100,130
```

Removes the breakpoints set before lines 100 and 130.

USING

Format

USING { *line-number* / *string-expression* }

Description

USING can be in a PRINT or DISPLAY statement to format the output. If *line-number* is given, the format is specified in that line by an IMAGE statement. *Line-number* must refer to a line in the current program or subprogram. See IMAGE. If *string-expression* is given, the format is defined by USING.

When USING is present, the following changes occur in the evaluation of the *print-list* of PRINT or DISPLAY.

- Comma print separators are treated as semicolons.
- The TAB function causes an error.
- The print items are formatted according to fields specified in the format definition. If the number of print items in *print-list* exceeds the number of fields in the format, the current formatted record is written. The remaining values are written in the next record, using the format definition again, from the beginning. The format is used as many times as is necessary to complete the *print-list*. A new record is generated each time the format is used. When the number of print items is less than the number of fields in the definition, output stops when the first field is encountered for which there is no print item.
- If a formatted item is too long for the remainder of the current record, it is divided into segments. The first segment fills the remainder of the current record and any remaining segments are written on the next record.

Cross Reference

DISPLAY, IMAGE, PRINT

VAL

Format

VAL(*string-expression*)

Description

The VAL function returns the numerical value of *string-expression*. Leading and trailing spaces are ignored. The VAL function is the inverse of the STR$ function.

If *string-expression* is not a valid representation of a number, an error occurs. To avoid this error, the *string-expression* may be checked first with the NUMERIC function.

Cross Reference

NUMERIC, STR$

Examples

```
170 NUM=VAL("78.6")
```

Sets NUM equal to 78.6.

```
190 LL=VAL("3E15")
```

Sets LL equal to 3.E + 15.

```
300 PRINT VAL("$3.50"):PAUSE
```

Causes an error because the string does not represent a valid numeric constant.

VERIFY

Format

VERIFY "*device.filename*" [, PROTECTED]

Description

The VERIFY command checks that data was saved on an external storage device or was loaded into memory correctly. VERIFY is used after a SAVE or OLD command to compare the program in memory to the program on the external storage device. If a difference is found, an error message is displayed. Both input/output errors 12 and 24 indicate a verification error.

Device.filename identifies the device and the file in which the program is stored. *Device* is the number associated with the physical device and can be from 1 through 255. *Filename* identifies the file.

Like SAVE, VERIFY removes any variable names which are not used in the program. If the program is protected, then PROTECTED must be specified in the VERIFY command.

Cross Reference

OLD, SAVE

Examples

```
SAVE "1.MYPROG"
```
Saves the file named MYPROG to device 1.
```
VERIFY "1.MYPROG"
```
Verifies whether the file was stored correctly.
```
OLD "1.STAT"
```
Reads the file named STAT into memory from device 1.
```
VERIFY "1.STAT"
```
Verifies whether the file was read correctly.

SUBPROGRAM

VERSION

Format

CALL VERSION(*numeric-variable*)

Description

The VERSION subprogram returns a value indicating the version of BASIC that is being used. The BASIC used on the CC-40 returns a value of 10.

Example

```
170 CALL VERSION(V)
```
Sets V equal to 10.