GENEVE

The

MYARC 9640

Family Computer

MYARC ADVANCED BASIC

User's Manual

MYARC, Inc.
Basking Ridge, NJ

COPYRIGHT

Copyright @1986 by MYARC, INC. All rights reserved. No part of this publication may be reproduced without the written permission of MYARC, INC., P.O. Box 140, Basking Ridge, NJ 07920

DISCLAIMER OF WARRANTY

MYARC, INC. makes no representation or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. MYARC, INC. software is sold or licensed "as is." The risk as to its quality and performance is with the buyer and not MYARC, INC. Further MYARC reserves the right to revise this publication and to make changes in the content hereof without obligations of MYARC to notify any person of such revisions or changes. MYARC also reserves the right to make design revisions or changes without obligations of MYARC to notify any person of such revisions or changes.

MYARC Advanced BASIC

This Manual contains an alphabetical listing of all MYARC Advanced BASIC commands, statements and functions with detailed explanations on each. The Appendix Section provides significant reference details that you will find necessary for effective programming.

MYARC Advanced BASIC is totally upward compatible with MYARC Extended BASIC II and with TI Extended BASIC so that you are already familiar with nearly all the referenced commands, statements, and functions.

In addition to many new commands, statements and functions that were not in MYARC Extended BASIC II, MYARC Advanced BASIC provides additional speed, power, flexibility, and/or sophistication.

Several MYARC Extended BASIC II commands and statements are no longer used in MYARC Advanced BASIC and many commands and statements that were used in MYARC Extended BASIC II have been revised and/or their descriptions modified to reflect the added flexibility that the User now will have and can take advantage of in MYARC Advanced BASIC.

To simplify I/O communication with external devices, a set of default I/O commands has been added to MYARC Advanced BASIC. These commands are described separately in the section "I/O Default Commands".

Different from MYARC Extended BASIC II and TI Extended BASIC, in MYARC Advanced BASIC the function "Break" is invoked by simultaneously depressing both the Control and Break keys. Accordingly wherever in this manual reference is made to "CLEAR", press the two keys, CONTROL + BREAK.

WE RECOMMEND THAT YOU CAREFULLY REVIEW THIS ENTIRE MANUAL BEFORE PROCEEDING WITH ANY SERIOUS PROGRAMMING.

TABLE OF CONTENTS

COMMANDS, STATEMENTS and FUNCTIONS

NOTES:

* Revised from MYARC Extended BASIC II.
** New commands, statements or functions.

ABS **ABS**

Format
ABS(numeric-expression)

Type
Numeric (REAL or DEFINT)

Description
The ABS function gives the absolute value of the numeric-expression.

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

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

ABS always returns a non-negative number.

Examples

```
100 PRINT ABS(45.2)
PRINT ABS(45.2)
Prints 45.2

100 VV=ABS(-7.345)
VV=ABS(-7.345)
Sets VV equal to 7.345
```

ACCEPT

ACCEPT

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

Cross Reference
GRAPHICS, INPUT, LINPUT, MARGINS, TERMCHAR, BCOLOR, BTIME

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

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

ACCEPT can be used as either a program statement or a command.

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

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

You normally press ENTER to complete keyboard input; however, you can also use Alt 7 (AID), Alt 9 (BACK), Alt 5 (BEGIN), CLEAR, Alt 6 (PROC'D), DOWN ARROW, or UP ARROW. You can use the TERMCHAR function to determine which of those keys was pressed to exit from the previous ACCEPT, INPUT, or LINPUT instruction.

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

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

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

BEEP--Sounds a short tone to signal that the computer is ready to accept input.

ERASE ALL--Places a space character (ASCII code 32) in every character position in the screen window before accepting input.

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

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

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

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

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

New Options

CLIP--Using the CLIP option, the string represented in the "DISPLAY AT" statement will be clipped at the end of a line rather than wrapping around to the next line, as it does in the default mode. The CLIP option is particularly useful when using "DISPLAY AT" within a window.

BLINK/INVERT--BLINK will cause the line displayed to BLINK on and off. This is only available in GRAPHICS(3,1) mode.

INVERT--Will cause the pixels in each character to invert their colors so the foreground- and background-colors will be inverted. This is only available in GRAPHICS(2,2), (2,3), (3,2), and (3,3) modes.

Examples

```
100 ACCEPT AT(3,5):Y
```
Accepts data at the third row, fifth column of the screen window into the variable Y.

```
100 ACCEPT VALIDATE("YN"):R$
```
Accepts data containing Y and/or N into the variable R$. (YYNN would be a valid entry.)

```
100 ACCEPT ERASE ALL:B
```
Accepts data into the variable B after putting the blank character into all positions in the screen window.

```
100 ACCEPT AT(R,C)SIZE(FIELDLEN)BEEP VALIDATE(DIGIT,"AYN"):X$
```
Accepts a digit or the letters A, Y, or N into the variable X$. The length of the input may be up to FIELDLEN characters. A field the length of FIELDLEN is filled with blank characters, and then the data value is accepted at row R, column C. A beep is sounded before acceptance of data.

Program

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

ASC **ASC**

Format
ASC(string-expression)

Cross Reference
CHR$

Description
The ASC function returns the ASCII character code corresponding to the first character of the string-expression.

ASC is the inverse of the CHR$ function.

The string-expression cannot be a null string.

Examples

```
100 PRINT ASC("A")
```
Prints 65 (the ASCII character code for the letter A).

```
100 B=ASC("1")
```
Sets B equal to 49 (the ASCII character code for the character 1).

```
100 DISPLAY ASC("HELLO")
```
Displays 72 (the ASCII character code for the letter H).

```
100 A$="DAVID"
110 PRINT ASC(A$)
```
Prints 68 in line 110.

ATN

ATN

Format
ATN(numeric-expression)

Cross Reference
COS, SIN, TAN

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

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

Examples

```
100 PRINT 4*ATN(-1)
```
Prints -3.141592654.

```
100 Q=PI/ATN(1.732)
```
Sets Q equal to 3.0000363894830.

BCOLOR

BCOLOR

Format
CALL BCOLOR(foreground,background)

Cross Reference
BTIME, DISPLAY, ACCEPT

Description
This command is used to set the foreground- and background-colors of the BLINK parameter used in conjunction with DISPLAY AT, ACCEPT AT and BTIME. The value of foreground- or background-color is 1 to 16 as given in Appendix F. This subroutine is applicable only to graphics 3,1 (Text 2) mode.

Example

```
100 CALL GRAPHICS(3,1)
110 CALL SCREEN(16,5)
120 CALL BCOLOR(16,7)
130 DISPLAY AT(5,1)ERASE ALL BLINK:"THIS IS BLINKING"
140 ACCEPT AT(5,1)BLINK SIZE(-28):A$
```

This program displays normal text in white with a dark blue background. The display area on line 5 will blink and alternately be white text on a dark red background and white text on a dark blue background.

BEEP **BEEP**

Cross Reference
DISPLAY AT, ACCEPT AT, SOUND

Description
The BEEP command sounds a short tone when encountered as a command or program statement. BEEP is also an option in DISPLAY AT and ACCEPT AT commands.

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

Example

```
100 CALL GRAPHICS(4)
110 DEFINT I,R,C
120 CALL POINT(1,R,C)
130 FOR I=1 TO 25
140 C=(RND5)+1
150 R=(RND1)+1
160 CALL DRAWTO(R,C)
170 BEEP
180 NEXT I
190 GOTO 190
(Press CLEAR to stop the program.)
```

This program randomly selects the ROW COLUMN coordinates of 25 points and draws lines connecting them sequentially. Each time a line is drawn the BEEP sound is produced.

BREAK

BREAK

Format
BREAK(line-number-list)

Cross Reference
CONTINUE, ON BREAK, UNBREAK

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

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

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

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

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

Breakpoints

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

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

It restores the default character definitions of all characters.

It restores the default foreground-color and background-color to all characters.

It restores the default screen color.

It deletes all sprites.

It resets the sprite magnification level to 1.

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

Removing Breakpoints

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

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

BREAK Errors

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

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

```
WARNING
LINE NOT FOUND
```

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

Examples

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

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

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

BTIME

BTIME

Format
CALL BTIME(blinkrate-ON, blinkrate-OFF)

Cross Reference
BCOLOR, ACCEPT, DISPLAY

Description
This command is used to set the rate at which characters are set to BLINK in the DISPLAY AT and ACCEPT AT statements.

Blinkrate can be an integer from 0 to 15, representing actual blink rates between 0 and 2503.5 milliseconds in multiples of 166.9 milliseconds.

Example

```
100 CALL GRAPHICS(3,1)
110 CALL DCOLOR(15,5)
120 CALL BCOLOR(15,7)
130 FOR I=0 TO 15
140 CALL BTIME(I,I)
150 DISPLAY AT(5,1)ERASE ALL BLINK:"RATE OF BLINK= ";I
160 FOR DELAY=1 TO 1000::NEXT DELAY
170 NEXT I
180 END
```

The above program illustrates some of the possible blink rates.

BYE

BYE

Format
BYE

Description
The BYE command resets the computer. Always use BYE to exit from MYARC Advanced BASIC. The BYE command causes the computer to do the following:

Close all open files.

Erase the program and all variable values in memory.

Exit from MYARC Advanced BASIC.

Display the DOS command line.

CALL **CALL**

Format
CALL subprogram-name[(parameter-list)]

Cross Reference
SUB

Description
The CALL instruction transfers program control to the specified subprogram.

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

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

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

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

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

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

CHAR	GCHAR	PEEKV
CHARPAT	GRAPHICS	POINT
CHARSET	HCHAR	POKEV
CIRCLE	INIT	POSITON
CLEAR	JOYST	RECTANGLE
COINC	KEY	SAY
COLOR	LINK	SCREEN
DCOLOR	LOAD	SOUND
DELSPRITE	LOCATE	SPGET
DISTANCE	MAGNIFY	SPRITE
DRAW	MARGIN	VCHAR
DRAWTO	MOTION	
ERR	PATTERN	
FILL	PEEK	

Program

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

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

CDBL CDBL

Format
(numeric-expression)

Cross Reference
DEFtype, CINT, CSNG, CREAL

Description
Converts a number to double-precision. The numeric-expression must evaluate to either an integer, or a single- or a double-precision value.

CAUTION: mixed mode arithmetic is not allowed.

Arithmetic modes:

REAL: Real numbers and integers.

BINARY: Integers, single-precision, double-precision.

Mixing real numbers with either single- or double-precision will cause a mixed arithmetic mode error.

CHAR -Subprogram

Format
CALL CHAR(character-code,pattern-string[,...])

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

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

CHAR is the inverse of the CHARPAT subprogram.

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

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

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

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

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

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

In Bit Mapped Modes, the displayed character remains unchanged.

Graphics(1,X) Modes

In Graphics(1,1), (1,2), and (1,3) modes, each character is composed of 64 pixels in a grid eight pixels high and eight pixels wide, as explained below.

You can use the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions and the HCHAR and VCHAR subprograms to display characters on the screen.

Other Graphics Modes

In Graphics(2,X) and (3,X), each character is composed of 48 pixels in a grid eight pixels high and six pixels wide. The eight by eight grid described below is used to define characters; however, the last two pixels in each pixel-row are ignored.

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

Character Definition--The Pattern String

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

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

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

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

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

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

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

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

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

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

Programs

For the dot pattern pictured below, you use "1898FF3D3C3CE404" as the pattern string for CALL CHAR. The following program uses this and one other string to make a figure "dance". This example will work only in Pattern Mode.

```
100 CALL CLEAR
110 A$="1898FF3D3C3CE404"
120 B$="1819FFBC3C3C2720"
130 CALL COLOR(27,7,12)
140 CALL VCHAR(12,16,244)
150 CALL CHAR(244,A$)
160 GOSUB 200
```

```
170 CALL CHAR(244,B$)
180 GOSUB 200
190 GOTO 150
200 FOR DELAY=1 TO 150
210 NEXT DELAY
220 RETURN
RUN
(screen clears)
(character moves)
(Press CLEAR to stop the program.)
```

To make this example work in a Bit-Mapped Graphics Mode, make the following changes.

```
105 CALL GRAPHICS(2,2)
130 CALL DCOLOR(7,12)
140 CALL CHAR(144,A$,145,B$)
150 CALL VCHAR(12,16,144)
170 CALL VCHAR(12,16,145)
```

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

The following example works in all graphics modes.

```
100 CALL CLEAR
110 CALL GRAPHICS(X,Y)
120 CALL CHAR(144,"FFFFFFFFFFFFFFFF")
130 CALL CHAR(42,"0F0F0F0F0F0F0F0F")
140 CALL HCHAR(12,17,42)
150 CALL VCHAR(14,17,144)
160 FOR DELAY=1 TO 500
170 NEXT DELAY
RUN
```

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

CHARPAT -Subprogram **CHARPAT**

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

Cross Reference
CHAR

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

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

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

See Appendix B for a list of available characters.

Example

```
100 CALL CHARPAT(33,C$)
```
Sets C$ equal to "0010101010001000", the pattern identifier for character 33, the exclamation point.

CHARSET -Subprogram--Set Characters **CHARSET**

Format
CALL CHARSET ([startchar][-][endchar])

Cross Reference
CHAR, COLOR, CCOLOR

Description
The CHARSET subprogram restores default character definitions and colors.

CHARSET, if not followed by (), restores the default character definitions to characters 32-95, inclusive.

In Graphics(1), CHARSET restores the default colors to all 256 characters.

If followed by parenthesis, CALL CHARSET(startchar-endchar) will restore only the specified characters.

Example

COMMAND	**ACTION**
CALL CHARSET	Restores characters 32 to 96 inclusive.
CALL CHARSET(X-Y)	Restores characters X to Y inclusive.
CALL CHARSET(-Y)	Restores all characters starting at 1 through Y inclusive.
CALL CHARSET(X-)	Restores all characters from X through 255 inclusive.

See Appendix B for a list of available characters.

CHR$** -Function--Character **CHR$

Format
CHR$(character-code)

Type
String

Cross Reference
ASC

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

CHR$ is the inverse of the ASC function.

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

Examples

```
100 PRINT CHR$(72)
```
Prints H.

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

Program

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

```
100 CALL CLEAR
110 IMAGE ### ## ### ##
120 FOR A=32 TO 127
130 PRINT USING 110:A,CHR$(A);
140 NEXT A
```

CINT **CINT**

Format
CINT(numeric-expression)

Cross Reference
DEFvartype, CDBL, CSNG, CREAL

Description
Converts a number to integer precision.

CAUTION: mixed mode arithmetic is not allowed.

Arithmetic modes:
REAL: real numbers and integers.

BINARY: integers, single-precision, double-precision.

Mixing real numbers with either single- or double-precision will cause a mixed arithmetic mode error.

CIRCLE -Subprogram **CIRCLE**

Format
CALL CIRCLE(x,y),radius,[,color[,start,end[,aspect ratio]]]

Description
Draws an ellipse on the screen with center at (x,y). A circle is drawn when the aspect ratio equals 1.

X,Y are relative coordinates of the screen. The coordinates 0,0 represent the extreme upper left hand corner of the screen.

X represents the vertical coordinate or the dot-column count to the right of the left hand edge of the screen.

Y represents the horizontal coordinate or the dot/row count from the top of the screen counting increasing towards the bottom.

Radius is the major axis of the ellipse.

Start, and end, are the beginning and ending angles in radians. Must be in the range of -2*PI to 2*PI.

Aspect ratio is the ratio of the X-radius to the Y-radius in terms of coordinates.

If aspect ratio is less than 1, radius is the x-radius and is measured in pixels in the horizontal direction.

If aspect is greater than 1, radius is the y-radius and is measured in pixels in the vertical direction.

When X-radius is equal to Y-radius, then aspect ratio is equal to 1 and a circle is drawn.

Example

```
CIRCLE(150,100),50
```

CLEAR -Subprogram **CLEAR**

Format
CALL CLEAR

Cross Reference
DCOLOR, DELSPRITE

Description
The CLEAR subprogram erases the screen.

CLEAR places a space character (ASCII code 32) in every screen position.

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

Programs

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

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

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

```
100 CALL CHAR(32,"0103070F1F3F7FFF")
110 CALL CLEAR
120 GOTO 120
RUN
--Screen is filled with *
(Press CLEAR to stop the program.)
```

The following program first fills and then clears the entire screen.

```
100 CALL GRAPHICS(1,2)
110 CALL HCHAR(1,2,72,768)
120 FOR DELAY=1 TO 500::NEXT DELAY
130 CALL CLEAR
140 GOTO 140
RUN
(Press CLEAR to stop the program.)
```

CLOSE

CLOSE

Format
CLOSE #file-number[:KILL]

Cross Reference
KILL, OPEN, DELETE

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

If #file-number is omitted, then basic will close all open files, however, the KILL option is not allowed without a specific #file-number.

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

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

The KILL option, which can be used only with certain devices, deletes the file after closing it. For more information about using the KILL option with a particular device, refer to the owner's manual that comes with that device.

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

Closing Files Without the CLOSE Instruction

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

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

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

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

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

Examples

Diskette File
```
100 OPEN #24:"DSK1.MYDATA",INTERNAL,UPDATE,FIXED
200 CLOSE #24
RUN
```
The CLOSE statement for a diskette requires no further action on your part.

CLS

CLS

Format
CLS

Description
You may use CLS either as a program statement or a command.

CLS clears the screen or active viewport created such as with the CALL MARGIN statement, and returns the cursor to the home position.

Examples

```
100 CALL GRAPHICS(2,1)
110 CALL MARGIN(1,24,1,40)
120 CALL HCHAR(1,1,ASC("A"),960)
130 CALL MARGIN(5,10,5,10)
140 CLS
150 GOTO 150
RUN
(Press CLEAR to stop the program.)
```

Program will fill the screen with character 65, the letter A, then it creates a window 5 rows by 5 columns. The CLS statement clears this window leaving the remainder of the screen filled with the letter "A".

NOTE: An alternate method of clearing the active "window" in this case would have been to substitute line 140 with:

```
140 DISPLAY AT(1,1)ERASE ALL:""
```

CALL CLEAR or CALL GRAPHICS(n) will clear the entire screen.

COINC -Subprogram--Coincidence **COINC**

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

Cross Reference
SPRITE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Program

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

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

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

COLOR --Subprogram **COLOR**

Format
Pattern Mode

```
CALL COLOR(character-set,foreground-color,background-color[,...])
```

Sprites

```
CALL COLOR(#sprite-number,foreground-color[,...])
```

Cross Reference
CHAR, DCOLOR, GRAPHICS, SCREEN, SPRITE

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

The types of parameters you specify in a call to the COLOR subprogram depend on whether you are assigning colors to characters or to sprites.

In general, each character has two colors. The color of the pixels that make up the character itself is the foreground-color; the color of the pixels that occupy the rest of the character position on the screen is the background-color.

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

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

See Appendix F for a listing of available colors and their respective codes.

Pattern Mode and Bit Mapped Modes

In these modes (i.e., Graphics(1,1), (2,2), (2,3), (3,2), (3,3)), the 256 available characters are divided into 32 sets of 8 characters each. When you assign a color combination to a particular set, you specify the colors of all 8 characters in that set.

> The character-set is a numeric-expression whose value specifies the number (0-31) of the 8-character set.
>
> Foreground-color and background-color are numeric/expressions whose values specify colors that can be assigned from among the 16 available colors.

See Appendix D for available characters and character sets in Pattern Mode.

Text Modes

An error occurs if you use the COLOR subprogram to assign character colors in either Text Mode (i.e., Graphics(2,1) or Graphics(3,1). Use the SCREEN subprogram to assign character colors in Text Mode. Sprites are not displayed in text mode.

Graphics(1,2) and (1,3)

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

Sprites

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

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

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

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

Examples

```
100 CALL COLOR(#5,16)
```
Sets sprite number 5 to have a foreground-color of 16 (white). The background-color is always 1 (transparent).

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

```
100 CALL COLOR(#7,INT(RND* 16+1))
```
Sets sprite number 7 to have a foreground-color chosen randomly from the 16 colors available. The background-color is 1 (transparent).

This example is valid in all graphics modes.

Program

This program sets the foreground-color of characters 48-55 to 5 (dark blue) and the background-color to 12 (light yellow).

```
100 CALL CLEAR
110 CALL GRAPHICS(1)
120 CALL COLOR(3,5,12)
130 DISPLAY AT(12,16):CHR$(48)
140 GOTO 140
(Press CLEAR to stop the program.)
```

CONTINUE **CONTINUE**

Format
CONTINUE
CON

Cross Reference
BREAK

Description
The CONTINUE command restarts a program which has been stopped by a breakpoint. It may be entered whenever a program has stopped running because of a breakpoint caused by the BREAK command or statement or pressing Control + Break keys (CLEAR.) However, you cannot use the CONTINUE command if you have edited a program line. CONTINUE may be abbreviated as CON.

When a breakpoint occurs, the standard character set and standard colors are restored. Sprites cease to exist. CONTINUE does not restore user-defined characters that have been reset or any colors. Otherwise, the program continues as if no breakpoint had occurred.

COS --Function--Cosine **COS**

Format
COS(numeric-expression)

Type
REAL

Cross Reference
ATN, SIN, TAN

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

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

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

Program

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

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

CREAL

CREAL

Format
CREAL(numeric-expression)

Cross Reference
DEFvartype, CDBL, CINT, CSNG

Description
Converts a number to single-precision.

CAUTION: mixed mode arithmetic is not allowed.

Arithmetic modes:
REAL: real numbers and integers.

BINARY: integers, single-precision, double-precision.

Mixing real numbers with either single- or double-precision will cause a mixed arithmetic mode error.

CSNG

CSNG

Format
CSNG (numeric-expression)

Cross Reference
DEFvartype, CINT, CDBL, CREAL

Description
Converts a number to single-precision.

CAUTION: mixed mode arithmetic is not allowed.

Arithmetic modes:
REAL: real numbers and integers.

BINARY: integers, single-precision, double-precision.

Mixing real numbers with either single- or double-precision will cause a mixed arithmetic mode error.

DATA

DATA

Format
DATA data-list

Cross Reference
READ, RESTORE

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

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

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

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

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

A DATA statement encountered during program execution is ignored.

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

Program

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

```
100 FOR A=1 TO 5
110 READ B,C
120 PRINT B;C
130 NEXT A
140 DATA 2,4,6,7,8
150 DATA 1,2,3,4,5
160 DATA """THIS HAS QUOTES"""
170 DATA NO QUOTES HERE
180 DATA " NO QUOTES HERE,EITHER"
190 FOR A=1 TO 6
200 READ B$
210 PRINT B$
220 NEXT A
230 DATA 1,NUMBER,MYARC
RUN
2 4
6 7
8 1
2 3
4 5
"THIS HAS QUOTES"
NO QUOTES HERE
  NO QUOTES HERE,EITHER
 1
NUMBER
MYARC
```

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

DATE/DATE$** **DATE/DATE$

Format
CALL DATE["mm/dd/yy"]
DATE$

Description
DATE$ can be a function.

CALL DATE can be a statement or a command.

It can be used to set the date or retrieve the current date.

To set the date use the format:
CALL DATE("mm/dd/yy")

mm is the two-digit equivalent of the current month 01 - 12
dd is the two digit date 01 - 31
yy is the last two digits of the year. Two-digit range = range 01 - 99.

To retrieve the current date, use the function DATE$.

Example

```
CALL DATE$("01/01/87")
```

This example sets the date to January 1, 1987.

Example

```
PRINT DATE$
01/01/87

100 A$=DATE$
110 PRINT A$
RUN
01/01/87
```

Example

```
100 PRINT "TODAY'S DATE IS ";DATE$
110 INPUT "DO YOU WISH TO CHANGE THE DATE ? ":CHANGE$
120 IF LEFT$(CHANGE$,1)="Y" OR LEFT$(CHANGE$,1)="y" THEN 130    ELSE END
130 INPUT "ENTER NEW DATE:":NEWDATE$
140 CALL DATE(NEWDATE$)
150 GOTO 100
```

DCOLOR --Subprogram--Draw Color **DCOLOR**

Format
CALL DCOLOR(foreground-color,background-color)

Cross Reference
CIRCLE, COLOR, DRAW, DRAWTO, FILL, GRAPHICS, HCHAR, POINT, RECTANGLE, VCHAR

Description
The DCOLOR subprogram enables you to set the graphics colors.

The graphics colors are used by the CIRCLE, DRAW, DRAWTO, FILL, HCHAR, POINT, RECTANGLE, and VCHAR subprograms in Bit Mapped Graphics and normal Graphics modes.

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

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

DCOLOR is effective only in Bit Mapped and normal Graphics modes. DCOLOR has no effect in Pattern or Text mode.

Programs

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

```
100 CALL CLEAR
110 CALL GRAPHICS(2,2)
120 CALL DCOLOR(5,8)
130 CALL HCHAR(8,20,72,3)
```

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

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

Line 130 changes the foreground-color (chosen randomly from the even-numbered colors available) and the background-color (chosen randomly from the odd-numbered colors).

DEF --Define Function **DEF**

Format
DEF function-name[(parameter1 [,. . . parameter7])]=expression

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

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

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

If the function-name is a numeric variable, you can optionally specify its data-type (DEFINT, DEFREAL, DEFSNG, or DEFDBL) by using variable tags.

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

If a parameter is a numeric variable, you can optionally specify its data-type (DEFINT, DEFREAL, DEFSNG, or DEFDBL) by using variable tags.

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

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

DEF Without Parameters

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

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

DEF With Parameters

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

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

Recursive Definitions

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

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

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

Examples

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

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

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

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

```
100 DEF FIRSTWORD$(NAME$)=SEG$(NAME$,1,POS(NAME$," ",1)-1)
```

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

DEFvartype **DEFvartype**

Vartypes: DEFINT, DEFSNG, DEFDBL, DEFREAL, DEFSTR

DEFINT - define as integers
DEFSNG - define as single-precision binary floating point (32 Bit)
DEFDBL - define as double-precision binary floating point (64 Bit)
DEFSTR - define as a string
DEFREAL - define as double-precision RADIX 99 floating point (64 Bit)

```
Format:  DEFINT I,J,COUNT,LOOPNUM,DIM A(100)
         DEFSNG X,Y,COST,DIM A(100)
         DEFDBL ANGLE,MEASUR,AN,DIM C(50)
         DEFREAL SQRROOT,VALUE,N,DIM D(40)
         DEFSTR NAM,FILENAME,N,F,DIM E(75)
```

NOTE: DEFREAL ALL is the default mode in MYARC Advanced BASIC.

Cross Reference
DIM, OPTION BASE, REAL, SUB

Description
The DEFvartype instruction enables you to declare the data-type of specified variables.

Usually the name given to a variable will identify the type of variable. Example: If a variable name ends in a dollar sign (ie., A$) then the variable is a string variable. Numeric variables can be identified in MYARC Advanced BASIC in terms of precision by the use of the following symbols as terminators attached to the end of the variable name. %, !, # or @ are termed type declaration tags. A numeric variable without such a tag is a double-precision variable in MYARC Advanced BASIC.

SYMBOL	TYPE OF VARIABLE
$	STRING VARIABLE
%	INTEGER CONSTANT
!	SINGLE PRECISION
#	DOUBLE PRECISION
@	REAL

Variables can also be declared by use of the DEFvartype statement. The declaration must be present and executable at a lower line number than that of any use of the variable-names that it represent.

A DEFvartype statement must appear at the beginning of a line. Also, any variable defined by that statement must appear later in the program.
part of an IF THEN statement.

The variable-list consists of one or more variables separated by commas. The DEFINT, DEFSNG, DEFDBL, and DEFREAL statements allow an ALL option, if this is used then all numeric variables in the program will be defined as the type specified except if they are specifically declared otherwise.

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

Integer variables are processed faster and use less memory that do real (or floating) point variables.

CAUTION: mixed mode floating point arithmetic is not allowed.

REAL: real numbers and integers.

BINARY: integers, single-precision, double-precision.

Mixing real numbers with either single- or double-precision will cause a mixed mode arithmetic error.

DEFvartype statements also can be used to declare the types of arrays.

TYPE-DECLARATION-TAGS override DEFvartype statements.

Programs

In the following example, DEFSTR NAM overrides DEFINT ALL such that NAM(5) will be treated as a string.

```
100 DEFINT ALL
110 DEFSTR NAM
120 NAM(5)="MYARC"::X%=37.123545:: I=1.2345
130 PRINT NAME;X;I
RUN
MYARC 37.12345 1
```

DELETE **DELETE**

Format
DELETE:[startline# - endline#]

Description
100-200 deletes lines 100-line 200 inclusive.

COMMAND	LINES DELETED
DELETE	All lines.
DELETE X	Line number X only.
DELETE X-	Lines from number X to the highest line number, inclusive.
DELETE -X	Lines from the lowest line number to line number X, inclusive.
DELETE X-Y or DELETE X,Y	All lines from line number X to line number Y, inclusive.

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

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

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

If line-number-range is between lines in the program, only those lines that fall within the range specified will be deleted.

NOTE: For TI 99/4A Programs:

Delete will no longer be used to delete files from DISK STORAGE DEVICE. See KILL, CLOSE, FILES. However, programs that contain a "DELETE" file statement will execute exactly as they did under TI BASIC or TI EXTENDED BASIC. The token used internally will now be occupied by the KILL command. As long as the program is stored in tokenized form (program file, or DV163 merge format), then execution will not be affected. On listing the program, the word "KILL" will be listed instead of "DELETE".

DELETE--no longer applies to files. DELETE applies to line numbers ONLY. To delete files, see KILL.

DELSPRITE --Subprogram--Delete Sprite **DELSPRITE**

Format
Delete Specified Sprite

```
CALL DELSPRITE(#sprite-number[,...])
```

Delete All Sprites

```
CALL DELSPRITE(ALL)
```

Cross Reference
CLEAR, SPRITE

Description
The DELSPRITE subprogram enables you to delete one or more sprites. All sprites are deleted when your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode.

Delete Specific Sprites

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

Delete All Sprites

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

Examples

```
100 CALL DELSPRITE(#3)
```
Deletes sprite number 3.

```
100 CALL DELSPRITE(#4,#3*C)
```
Deletes sprite number 4 and the sprite whose number is found by multiplying 3 by C.

```
100 CALL DELSPRITE(ALL)
```
Deletes all sprites.

DIM --Dimension DIM

Format
DIM array-name(integer1[,... integer7])[,array-name...]

Cross Reference
OPTION BASE

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

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

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

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

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

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

You can optionally specify the data-type (DEFvartype) of a numeric array by replacing DIM with the data-type.

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

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

You cannot use OPTION BASE as a command.

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

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

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

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

Referencing an Array

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

Reserving Space for Arrays

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

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

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

Naming Arrays

The rules for naming array variables follow the same pattern as the rules for other type variables, namely if a variable name ends in variable type descriptor then the descriptor defines the variable type.

NOTE: if a DEFSTR statement is executed then a string array name need not end in a $.

Array variable names ending in % ! # or @ respectively refer to integer, single-precision, double-precision variables or real variables.

Type/declaration tags such as $, %, !, # or @ take precedence over DEFvartype all declarations.

The following statements will remove arrays from memory:

NEW, OLD, MERGE, RUN (without continue)

CALL MEMSET --sets all elements of an array to a defined value.

Examples
100 DIM X$(30)
Reserves space in the computer's memory for 31 string members of the array called X$.

100 DIM D(100),B(10,9)
Reserves space in the computer's memory for 101 members of the array called D and 110 (11 times 10) members of the array called B.

DISPLAY **DISPLAY**

Format
DISPLAY [print-list]
DISPLAY [AT(row,column)] [BEEP] [ERASE ALL][CLIP][INVERSE/BLINK]
[SIZE(numeric-expression)][:print-list]

Cross Reference
DISPLAY USING, GRAPHICS, MARGIN, PRINT, BTIME, BCOLOR

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

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

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

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

Options

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

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

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

ERASE ALL--The ERASE ALL option places a space character (ASCII code 32) in every character position in the screen window before displaying the data.

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

New Options

CLIP--Using the CLIP option, the string represented in the "DISPLAY AT" statement will be clipped at the end of a line rather than wrapping around to the next line, as it does in the default mode. The CLIP option is particularly useful when using "DISPLAY AT" within a window.

BLINK/INVERT--BLINK will cause the line displayed to BLINK on and off. This is only available in GRAPHICS(3,1) mode.

INVERT--Will cause the pixels in each character to invert their colors so the foreground- and background-colors will be inverted. This is only available in GRAPHICS(2,2), (2,3), (3,2), (3,3) modes.

Examples

```
100 DISPLAY AT(5,7):Y
```
Displays the value of Y at the fifth row, seventh column of the screen. It first clears row 5 from column 7 to the right margin.

```
100 DISPLAY ERASE ALL:B
```
Puts the blank character into all positions within the current screen window before displaying the value of B.

```
100 DISPLAY AT(R,C) SIZE(FIELDLEN)BEEP:X$
```
Displays the value of X$ at row R, column C. First it beeps and blanks FIELDLEN characters.

Program

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

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

This example is valid only in Pattern Mode.

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

DISPLAY USING **DISPLAY USING**

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

Cross Reference
DISPLAY, IMAGE, PRINT

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

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

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

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

See DISPLAY under "Options" for an explanation of the options AT, BEEP, ERASE ALL, and SIZE.

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

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

You cannot use the TAB function.

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

Examples

```
100 N=23.43
110 DISPLAY AT(10,4):USING"##.##":N
```
Displays the value of N at the tenth row and fourth column, with the format "##.##", after first clearing row 10 from column 4 to the right margin.

```
100 DISPLAY USING "##.##":N
```
Displays the value of N at the 24th row and first column, with the format "##.##".

DISTANCE --Subprogram **DISTANCE**

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

Cross Reference
COINC, SPRITE

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

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

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

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

Two Sprites

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

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

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

A Sprite and a Screen Pixel

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

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

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

The number returned to the numeric-variable equals the square of the distance between the sprite and the screen pixel.

Examples

```
100 CALL DISTANCE(#3,#4,DIST)
```
Sets DIST equal to the square of the distance between the upper-left corners of sprite #3 and sprite #4.

```
100 CALL DISTANCE(#4,18,89,D)
```
Sets D equal to the square of the distance between the upper-left corner of sprite #4 and position 18,89.

DRAW --Subprogram DRAW

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

Cross Reference
CIRCLE, DCOLOR, DRAWTO, FILL, GRAPHICS, POINT, RECTANGLE

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

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

TYPE	ACTION
1	Draws a line of the foreground-color specified by the DCOLOR subprogram. This is accomplished by turning on each pixel in the specified line.
0	Erases a line. This is accomplished by turning off each pixel in the specified line.
2	Reverses the status of each pixel on the specified line. (If a pixel is on, it is turned off; if a pixel is off, it is turned on.) This effectively reverses the color of the specified line.

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

Pixel-row must have a value from 1 to 192. Pixel-column must have a value from 1 to 256.

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

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

DRAW cannot be used in Pattern or Text modes of display. An error results if you use DRAW in Pattern or Text Modes.

In Graphics(1,2) and (1,3) modes, the computer divides each pixel-row into 32 groups of 8 pixels each. (This is most obvious when you assign a background-color other than cyan or transparent.) The computer can assign 1 foreground-color and 1 background-color, from among the 16 available colors, to each 8-pixel group.

In the Bit-Mapped modes, each pixel is independent of every other pixel on the screen.

Programs

The following program draws a large triangle on the right of the screen.

```
100 CALL GRAPHICS(3)
110 CALL CLEAR
120 CALL DRAW(1,19,185,97,115)
130 CALL DRAW(1,19,185,97,255)
140 CALL DRAW(1,97,115,97,255)
150 GOTO 150
(Press CLEAR to stop the program.)
```

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

```
100 CALL CLEAR
110 CALL GRAPHICS(3)
120 CALL SCREEN(6)
130 FOR X=1 TO 255 STEP 5
140 CALL DRAW(1,1,X,128,256-X)
150 NEXT X
160 GOTO 160
(Press CLEAR to stop the program.)
```

DRAWTO --Subprogram **DRAWTO**

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

Cross Reference
CIRCLE, DCOLOR, DRAW, FILL, GRAPHICS, POINT, RECTANGLE

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

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

TYPE	ACTION
1	Draws a line of the foreground-color specified by the DCOLOR subprogram. This is accomplished by turning on each pixel in the specified line.
0	Erases a line. This is accomplished by turning off each pixel in the specified line.
2	Reverses the status of each pixel on the specified line. (If a pixel is on, it is turned off; if a pixel is off, it is turned on.) This effectively reverses the color of the specified line.

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

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

Pixel-row must have a value from 1 to 192, pixel-column must have a value from 1 to 256.

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

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

DRAWTO cannot be used in Pattern or Text modes of display. An error results if you use DRAWTO in Pattern or Text Modes.

In Graphics(1,2) and (1,3) modes, the computer divides each pixel-row into 32 groups of 8 pixels each. (This is most obvious when you assign a background-color other than cyan or transparent.) The computer can assign 1 foreground-color and 1 background-color (from among the 16 available colors), to each 8-pixel group.

Program

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

```
100 CALL GRAPHICS(3)
110 CALL CLEAR
120 A=20::B=20
130 CALL DRAW(1,A,B,A,B)
140 FOR X=1 TO 10
150 B=B+20
160 CALL DRAWTO(1,A,B)
170 CALL DRAWTO(1,A+20,B-20)
180 CALL DRAWTO(1,A+20,B)
190 CALL DRAWTO(1,A,B-20)
200 NEXT X
210 GOTO 210
(Press CLEAR to stop the program.)
```

END **END**

Format
END

Cross Reference
STOP

Description
The END statement stops the execution of your program.

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

It closes all open files.

It restores the default character definitions of all characters.

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

It restores the default screen color (cyan).

It deletes all sprites.

It resets the sprite magnification level to 1.

The graphics colors (see DCOLOR) and current position (see DRAWTO) are not affected.

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

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

EOF EOF

Format
EOF(file-number)

Type
DEFINT

Cross Reference
ON ERROR

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

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

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

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

The EOF function cannot be used with an audio cassette.

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

Examples

```
100 PRINT EOF(3)
```
Prints a value according to whether you are at the end of the file opened as #3.

```
100 IF EOF(27)<>0 THEN 1150
```
Transfers control to line 1150 if you are at the end of the file opened as #27.

```
100 IF EOF(27) THEN 1150
```
Transfers control to line 1150 if you are at the end of the file opened as #27.

ERR --Subprogram--Error ERR

Format
CALL ERR(error-code,error-type[,error-severity,[line-number]])

Cross Reference
ON ERROR

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

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

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

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

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

ERR returns a two- or three-digit number to the numeric variable error-code. See Appendix J for a list of error codes and the conditions that cause them to be displayed.

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

An error-code of 0 indicates that no error has occurred.

The error-type is a numeric variable.

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

A negative error-type indicates that the error occurred during program execution.

An error-type of 0 indicates that no error has occurred.

Options

The value returned to the numeric variable error-severity is always nine.

The value returned to the numeric variable line-number is the line number of the program statement that was executing when the error occurred.

Examples

```
100 CALL ERR(A,B)
```
Sets A equal to the error-code and B equal to the error-type of the most recent error.

```
100 CALL ERR(W,X,Y,Z)
```
Sets W equal to the error-code, X equal to the error-type, Y equal to the error-severity, and Z equal to the line-number of the most recent error.

Program

The following program illustrates a use of CALL ERR.

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

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

EXP --Function--Exponential **EXP**

Format
EXP(numeric-expression)

Type
REAL

Cross Reference
LOG

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

EXP is the inverse of the LOG function.

The value of e is 2.718281828459.

Examples

```
100 Y=EXP(7)
```
Assigns to Y the value of e raised to the seventh power, which yields 1096.6331584290.

```
100 L=EXP(4.394960467)
```
Assigns to L the value of e raised to the 4.394960467 power, which yields 81.0414268887.

FILES

FILES

Format
CALL FILES(pathname)

Cross Reference
DOS Manual, Pathnames, Directories, OPEN, CLOSE, KILL

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

Displays the names of the files and directories on a disk. If pathname is specified, BASIC lists all files that match that pathname. Default is all files and directories in the current directory on the current drive.

To halt listing, depress any key. To continue the listing, press another, or the same key.

Examples

CALL FILES("DSK1")
Displays files in drive 1.

CALL FILES("RD")
Displays files on RD (ramdisk).

CALL FILES("DSK.UTILITIES")
Searches all drives for the disk named "UTILITIES" and displays files.

FILL --Subprogram FILL

Format
CALL FILL(pixel-row,pixel-column)

Cross Reference
CIRCLE, DCOLOR, DRAW, DRAWTO, GRAPHICS, POINT, RECTANGLE

Description
The FILL subprogram enables you to fill in the area surrounding a specified pixel with a specified color.

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

Character-code is a numeric-expression with a value from 0-215 specifying the character with which to fill the area surrounding the specified pixel.

Pixel-row must have a value from 1 to 192, pixel-column must have a value from 1 to 256.

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

The area surrounding the specified pixel is filled with the fill pattern until a screen edge or a foreground pixel (a pixel that is turned on) is encountered.

The boundaries of the area to be filled can be defined by lines drawn with the CIRCLE, DRAW, DRAWTO, POINT, RECTANGLE subprograms.

FILL cannot be used in Pattern or Text modes. An error results if you use FILL in Pattern or Text Modes.

In Graphics(1,2) and (1,3) modes the computer divides each pixel-row into 32 groups of 8 pixels each. The computer can assign a foreground-color and a background-color (from among the 16 available colors) to each 8-pixel group.

Program

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

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

FOR TO

FOR TO

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

Cross Reference
NEXT

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

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

FOR-NEXT Loop Execution

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

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

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

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

Specifications

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

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

The initial-value is a numeric-expression.

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

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

Nested FOR-NEXT Loops

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

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

Each nested loop must use a different control-variable.

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

FOR TO as a Program Statement

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

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

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

FOR TO as a Command

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

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

Examples

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

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

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

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

Program

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

```
100 CALL CLEAR
110 D=0
120 FOR CHAR=33 TO 63 STEP 30
130 FOR ROW=1+D TO 21+D STEP 4
140 FOR COLUMN=1+D TO 29+D STEP 4
150 CALL VCHAR(ROW,COLUMN,CHAR)
160 NEXT COLUMN
170 NEXT ROW
180 D=2
190 NEXT CHAR
200 GOTO 200
```

(Press CLEAR to stop the program.)

FREESPACE --Function **FREESPACE**

Format
FREESPACE

Type
REAL

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

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

Garbage Collection

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

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

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

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

Example

PRINT FREESPACE
Prints a value that indicates the amount of available memory.

GCHAR --Subprogram--Get Character **GCHAR**

Format
Pattern and Text Modes
```
     CALL GCHAR(row,column,numeric-variable)
```
High-Resolution Mode
```
     CALL GCHAR(pixel-row,pixel-column,numeric-variable)
```

Cross Reference
GRAPHICS, HCHAR, VCHAR

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

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

Pattern and Text Modes

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

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

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

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

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

High-Resolution Mode

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

The value of the numeric-expression pixel-row and -column must be greater than or equal to 1. In High-Resolution Mode, pixel-row must be less than or equal to 192. See Appendix K for Graphics Modes ranges.

The value of the numeric-expression pixel-column must be greater than or equal to 1 and less than or equal to the value of pixel columns on screen. See Appendix K.

The color of the specified screen pixel is given by the value returned to the numeric-variable.

Examples

```
100 CALL GCHAR(12,16,X)
```
Assigns to X the ASCII code of the character at row 12, column 16 in Pattern and Text modes.

```
100 CALL GCHAR(R,C,K)
```
Assigns to K the ASCII code of the character that is in row R, column C in Pattern and Text modes.

GOSUB --Go to a Subroutine **GOSUB**

Format
GOSUB line-number
GO SUB

Cross Reference
ON GOSUB, RETURN

Description
The GOSUB statement transfers program control to the specified subroutine. A subroutine frequently is used to perform a specific operation several times in the same program.

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

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

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

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

Nested Subroutines

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

For example, you might use GOSUB in your main program to transfer control to a subroutine. When the computer encounters a RETURN in the second subroutine the GOSUB in the first subroutine. Then, when a RETURN is encountered in the first subroutine, program control returns to the statement following the GOSUB in your main program.

Example

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

Program

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

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

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

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

GOTO GOTO

Format
GOTO line-number
GO TO

Cross Reference
ON GOTO

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

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

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

Program

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

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

GRAPHICS --Subprogram **GRAPHICS**

Format
CALL GRAPHICS(graphics-mode1, graphics-mode2)

Cross Reference
CHAR, CIRCLE, COLOR, DCOLOR, DRAW, DRAWTO, FILL, MARGIN, POINT, RECTANGLE, SCREEN

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

Graphics-mode is defined by a pair of numbers, the first of which defines the screen width (i.e., 1=32 characters, 2=40 characters, 3=80 characters). the second defines the mode the display is currently operating at (i.e., text or bit-mapped). See Appendix K for a more detailed description of each graphics mode.

When you enter MYARC Advanced BASIC, the computer is in Pattern Mode.

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

Clears the entire screen.

Restores the default character definitions of all characters.

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

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

Restores the default screen color (cyan).

Deletes all sprites.

Resets all sprites.

Resets the sprite magnification level to 1.

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

Turns off all sound.

Pattern Mode

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

In Pattern Mode, you have access to sprites.

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

Text Modes

In Text Modes, the screen is considered to be a grid 24 characters high and 40 characters wide (Graphics(2,1)) or 26 c_ h_ and 80 c_ w_ (Graphics(3,1)). Each character is 8 pixels high and 6 pixels wide.

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

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

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

Graphics(1,2) and (1,3)

In these modes, the screen is considered to be a grid 192 pixels high and 256 pixels wide.

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

Use the COLOR subprogram only to assign colors to sprites; any other use of the COLOR subprogram causes an error.

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

Use the COLOR subprogram only to assign colors to sprites; any other use of the COLOR subprogram causes an error.

In these modes, you have access to sprites.

In order to maintain compatibility with MYARC Extended BASIC II, CALL GRAPHICS(1), (2), and (3) will be supported as follows:

```
CALL GRAPHICS (1) = CALL GRAPHICS(1,1)
CALL GRAPHICS (2) = CALL GRAPHICS(2,1)
CALL GRAPHICS (3) = CALL GRAPHICS(1,2)
```

All programs using these older calls to graphics will run with no modification.

In these modes the computer divides each pixel-row into 32 groups of 8 pixels. The computer can assign a foreground-color and a background-color (from among the 16 available colors) to each 8-pixel group.

Bit-Mapped Graphics Modes

In bit-mapped graphics modes, each pixel on the screen is totally independent from any other. Each character of text is 8 pixels high and 6 pixels wide.

Graphics Mode	**Screen Dimension (Pixel)**	**Screen Dimension (Text)**
2,2	256 x 212	40 x 26
2,3	256 x 212	40 x 26
3,2	512 x 212	80 x 26
3,3	512 x 212	80 x 26

Example

```
100 CALL GRAPHICS (3)
```

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

HCHAR --Subprogram--Horizontal Character **HCHAR**

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

Cross Reference
DCOLOR, GCHAR, GRAPHICS, VCHAR

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

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

The value of row must be greater than or equal to 1, and must be less than or equal to the total number of rows available on the screen. The value of column must be greater than or equal to 1 and must be less than or equal to the total number of columns available on the screen.

HCHAR is not affected by margin settings.

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

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

If you use HCHAR to display a character on the screen, and then later use CHAR, COLOR, or DCOLOR to change the appearance of that character, the result depends on the Graphics Mode.

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

In other modes the displayed character remains unchanged.

Examples
```
100 CALL HCHAR(12,16,33)
```
Places character 33 (an exclamation point) in row 12, column 16.

```
100 CALL HCHAR(1,1,ASC("!"),768)
```
Places an exclamation point in row 1, column 1, and repeats it 768 times, which fills the screen in Pattern Mode.

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

HEX$** **HEX$

Format
HEX$(numeric-expression)

Description
Returns hexadecimal string equivalent to numeric-expression.

This command functions on integer values only.

Example

```
A$ = HEX$(-1)::PRINT A$
```
The computer prints:
FFFF

IF THEN ELSE

IF THEN ELSE

Format
```
IF relational-expression THEN line-number1 [ELSE line-number2]
   numeric-expression        statement1        statement2
```

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

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

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

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

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

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

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

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

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

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

IF THEN is followed by a statement, program execution proceeds with the next program line.

IF THEN is followed by a line-number only, program execution proceeds with the next program line.

IF THEN is followed by a line-number and a statement separator, program execution proceeds with the statements after the statement separator. Note: In this case, the statement separator symbol functions as an implied ELSE.

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

Examples

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

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

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

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

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

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

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

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

Program

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

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

IMAGE IMAGE

Format
IMAGE format-string

Cross Reference
DISPLAY USING, PRINT USING

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

The format-string is a string constant.

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

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

An IMAGE statement is not executed.

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

Format-Fields

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

A format-field may consist of the following characters:

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

To indicate that a number is to be given in scientific notation, circumflexes (^) must be given for the E and power numbers. There must be four or five circumflexes, and 10 or fewer characters (minus sign, pound signs, and decimal point) when using the E format.

The decimal point separates the whole and fractional portions of numbers, and is printed where it appears in the IMAGE statement.

All other letters, numbers, and characters are printed exactly as they appear in the IMAGE statement.

Format-string may be enclosed in quotation marks. If it is not enclosed in quotation marks, leading and trailing spaces are ignored. However, when used directly in PRINT...USING or DISPLAY...USING, it must be enclosed in quotation marks.

Each IMAGE statement may have space for many images, separated by any character except a decimal point. If more values are given in the PRINT USING or DISPLAY USING statement than there are images, then the images are reused, starting at the beginning of the statement.

If you wish, you may put format-string directly in the PRINT...USING or DISPLAY USING statement immediately following USING. However, if a format-string is used often, it is more efficient to refer to an IMAGE statement.

Examples

```
100 IMAGE $####.###
110 PRINT USING 100:A
```

IMAGE $####.### allows printing of any number from -999.999 to 9999.999. The following illustrates how some sample values would be printed or displayed:

VALUE	APPEARANCE
-999.999	$-999.999
-34.5	$ -34.500
0	$ 0.000
12.4565	$ 12.457
6312.991	$6312.999
99999999	$********

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

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

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

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

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

VALUES	APPEARANCE
JOHN	DEAR JOHN,
TOM	DEAR TOM ,
RALPH	DEAR ****,

Programs

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

```
100 CALL CLEAR
110 IMAGE $####.##
120 IMAGE " ####.##"
130 DATA 233.45,-147.95,8.4,37.263,-51.299,85.2,464
140 TOTAL=0
150 FOR A=1 TO 7
160 READ AMOUNT
170 TOTAL=TOTAL+AMOUNT
180 IF A=1 THEN PRINT USING 110:AMOUNT ELSE PRINT USING 120:AMOUNT
190 NEXT A
200 PRINT "-------"
210 PRINT USING "$####.##":TOTAL
RUN
$ 233.45
 -147.95
    8.40
   37.26
  -51.30
   85.20
  464.00
-------
$ 629.06
```

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

Line 180 prints the values using the IMAGE statements.

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

The amounts are printed with the decimal points aligned.

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

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

INIT --Subprogram--Initialize **INIT**

Format
CALL INIT[(numeric-expression)]

Cross Reference
LINK, LOAD

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

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

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

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

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

The maximum amount of memory space that you can allocate by using INIT is 49,152 bytes.

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

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

Examples

CALL INIT
Allocates 8K bytes of memory space.

CALL INIT(200)
Allocates 200 bytes of memory space.

CALL INIT(0)
Releases all memory previously allocated.

INPUT **INPUT**

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

Cross Reference
ACCEPT, EOF, LINPUT, OPEN, REC, TERMCHAR

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

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

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

Input from the Keyboard

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

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

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

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

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

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

A string value entered from the keyboard can optionally be enclosed in quotation marks. However, a string containing a comma, a quotation mark, or

leading or trailing spaces must be enclosed in quotation marks. A quotation mark within a string is represented by two adjacent quotation marks.

You normally press ENTER to complete keyboard input; however, you can also use Alt 7(AID), Alt 9(BACK), Alt 5(BEGIN), CLEAR, Alt 6(PROC'D), DOWN ARROW, or UP ARROW. You can use the TERMCHAR function to determine which of these keys was pressed to exit from the previous INPUT, LINPUT, or ACCEPT instruction.

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

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

Examples

```
100 INPUT X
```
Allows the input of a number.

```
100 INPUT X$,Y
```
Allows the input of a string and a number.

```
100 INPUT "ENTER TWO NUMBERS: ":A,B
```
Displays the prompt ENTER TWO NUMBERS and then allows the entry of two numbers.

```
100 INPUT A(J),J
```
First evaluates the subscript of A and then accepts data into that element of the array A. Then a value is accepted into J.

```
100 INPUT J,A(J)
```
First accepts data into J and then accepts data into the Jth element of the array A.

Program

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

```
100 CALL CLEAR
110 INPUT "ENTER YOUR FIRST NAME: ":FNAME$
120 INPUT "ENTER YOUR LAST NAME: ":LNAME$
130 INPUT "ENTER A THREE DIGIT NUMBER: ":DOLLARS
140 INPUT "ENTER A TWO DIGIT NUMBER: ":CENTS
150 IMAGE OF $###.## AND THAT IF YOU
160 CALL CLEAR
170 PRINT "DEAR ";FNAME$;",": :
180 PRINT "     THIS IS TO REMIND YOU"
190 PRINT "THAT YOU OWE US THE AMOUNT"
200 PRINT USING 150:DOLLARS+CENTS/100
210 PRINT "IF YOU DO NOT PAY US, YOU WILL SOON"
220 PRINT "RECEIVE A LETTER FROM OUR"
```

```
230 PRINT "ATTORNEY, ADDRESSED TO"
240 PRINT FNAME$;" ";LNAME$;"!": :
250 PRINT TAB(15);"SINCERELY,": : :TAB(15);"I.  DUN YOU": : : :
260 GOTO 260
(Press CLEAR to stop the program.)
```

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

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

Input from a File

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

> The file-number is a numeric-expression whose value specifies the number of the file as assigned in its OPEN instruction.
>
> If necessary, file-number is rounded to the nearest integer.
>
> If you use the REC option, the record-number is a numeric-expression whose value specifies the number of the record from which you want to input to the variable-list. The records in a file are numbered sequentially, starting with zero. The REC option can be used only with a file opened for RELATIVE access.
>
> If necessary, record-number is rounded to the nearest integer.

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

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

> In the case of INTERNAL FIXED records, null strings are assigned to the remaining variables, causing a program error if any of the remaining variables are numeric.
>
> For other records, the computer reads the next record in the file, and uses its values to complete the variable-list.

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

Examples

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

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

```
100 INPUT #11,REC 44:TAX
```
Puts into TAX the first value of record number 44 of the file that was opened as #11 with RELATIVE file organization.

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

Program

The following program illustrates a use of the INPUT statement. It opens a file on disk drive 1 called TEST and writes 5 records to the file. It then goes back and reads the records and displays them on the screen.

```
100 OPEN #1:"DSK1.TEST",SEQUENTIAL,INTERNAL,OUTPUT,FIXED 64
110 FOR A=1 TO 5
120 PRINT #1:"THIS IS RECORD",A
130 NEXT A
140 CLOSE #1
150 CALL CLEAR
160 OPEN #1:"DSK1.TEST",SEQUENTIAL,INTERNAL,INPUT,FIXED 64
170 PRINT
180 FOR B=1 TO 5
190 INPUT #1:A$,C
200 PRINT A$;C
210 NEXT B
220 CLOSE #1
RUN

THIS IS RECORD 1
THIS IS RECORD 2
THIS IS RECORD 3
THIS IS RECORD 4
THIS IS RECORD 5
```

INT --Function--Integer **INT**

Format
INT(numeric-expression)

Type
Real

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

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

Examples

```
100 PRINT INT(3.4)
```
Prints 3.

```
100 X=INT(3.9)
```
Sets X equal to 3.

```
100 P=INT(3.9999999999)
```
Sets P equal to 3.

```
100 DISPLAY AT(3,7):INT(4.0)
```
Displays 4 at the third row, seventh column of the current screen window.

```
100 N=INT(-3.9)
```
Sets N equal to -4.

```
100 K=INT(-3.00000001)
```
Sets K equal to -4.

JOYST --Subprogram--Joystick **JOYST**

Format
CALL JOYST(key-unit,x,y)

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

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

The position of the specifed joystick is returned in the numeric variables x and y as follows:

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

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

Example

```
100 CALL JOYST(1,X,Y)
```
Returns values in X and Y according to the position of joystick number 1.

Program

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

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

Expanded usage of the KEY command has been incorporated into the MYARC 9640.

Using the familiar command CALL KEY, the KEY subprogram is invoked. This KEY subprogram has been enlarged to also cover MYARC Advanced BASIC.

In addition, using the newly added KEY (not CALL KEY) commands, you can now change or tailor the functions performed by individual program function keys in various ways to accomodate your own programming needs. Three different constructs are used to change and/or utilize your redefined keys.

CALL KEY --subprogram

Format
CALL KEY(key-unit,key,status)

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

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

The numeric-expression key-unit can have a value from 0 to 6, as explained below.

The character code of the key pressed is returned in the numeric variable key. If no key is pressed, a value of 0 is returned.

See Appendix B for a list of the available characters.

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

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

Key-Unit Options

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

KEY-UNIT	RESULT
0	Console keyboard, in mode previously specified by CALL KEY.
1	Only the left side of the keyboard is active.
2	Only the right side of the keyboard is active.

3 Places keyboard in standard TI 99/4 mode. (Most Command Module software use this mode.) Both upper- and lower-case alphabetical characters are returned by the computer as upper-case only, and the function keys (BACK, BEGIN, etc.) return codes 1 through 15. No control characters are active.

4 Remaps the keyboard in the PASCAL mode. Both upper- and lower-case alphabetical character codes are returned by the computer, and the function keys return codes from 129 through 143. The control character codes are 1 through 31.

5 Places the keyboard in 99/4A BASIC mode. Both upper- and lower-case alphabetical character codes are returned by the computer. The function key codes are 1 through 15, and the control key codes are 128 through 159 (and 187.)

6 Places the keyboard in MYARC Advanced BASIC mode. See Appendix M, Additional Extended ASCII Codes for Keyboard Mode 6.

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

-1 The same key was pressed as was returned the last time KEY was called.

0 No key was pressed.

1 A different key was pressed than was returned the last time KEY was called.

Example

```
100 CALL KEY(0,K,S)
```
Returns in K the ASCII code of any key pressed on the keyboard except SHIFT, CTRL, FCTN, and CAPS, and in S a value indicating whether a key was pressed.

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

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

```
100 CALL CLEAR
110 CALL SPRITE(#1,33,5,96,128)
120 CALL KEY(1,K,S)
130 IF S=0 THEN 120
140 IF K=5 THEN Y=-4
```

```
150 IF K=0 THEN Y=4 160 IF K=2 THEN Y=-4
170 IF K=3 THEN X=4
180 IF K=1 THEN X,Y=0
190 IF K>5 THEN X,Y=0
200 CALL MOTION(#1,Y,X)
210 GOTO 120
(Press CLEAR to stop the program.)
```

KEY COMMANDS FOR REDEFINING FUNCTION KEYS

KEY

Format
KEY numeric expression, string expression

Description
The KEY numeric expression, string expression command allows you to redefine the associated string of a specified function key. Upon invoking BASIC, each function key has an associated string function as given in TABLE 1, pages 17 and 19 in the MYARC 9640 Manual. The purpose of this command is to allow you to redefine the default for any specified function key.

> Numeric expression defines the function key number that is being redefined. Valid function key numbers are 1-16.
> (Note: most present-day keyboards have 10 function keys.)
>
> String expression defines the string that is to be returned when the function key is pressed.

Either in the imperative mode (cursor blinking), or when a program is asking for input while running, pressing the function key will return its associated string.

Format
ON KEY (numeric expression) GOSUB line number

KEY(numeric expression) ON/OFF

Description
The ON KEY (numeric expression) GOSUB line number and the KEY(numeric expression) ON/OFF commands enable a running program to be halted and execution transferred to a predefined subprogram when a function key is pressed.

To successfully allow the program to transfer to the desired subroutine, you must first tell MYARC Advanced BASIC which function key is to transfer control to where, and also to tell the basic interpreter and operating system to start looking for the occurrence of a function key depression.

The numeric expression must be a valid function key number from 1 to 16. The line number tells the basic interpreter where the subroutine is to start once

the function key is pressed. However, you must also inform the operating system to look for, and to tell the basic interpreter that a particular function key has been pressed. This is performed by KEY(numeric expression) ON command.

When this statement is executed after an ON KEY (numeric expression) GOSUB line number for that particular function key, the function key is now considered "armed". Once armed, the basic interpreter will check with the operating system at the end of every line in the program, to determine if the function key has been pressed.

If yes, program control will be transferred to the line number given, just as if the main program had executed a GOSUB. The subroutine can end in either a RETURN, to return control to the program where it had been before being interrupted or, to a RETURN line number where execution returns to the line number given. In either case program execution will continue as if nothing had happened.

However, there are two cases when an armed function key is pressed but execution will not be interrupted:

1. If the program is executing the subroutine of a previously pressed function key and a RETURN has not yet occurred.

2. If you are in a user-defined subprogram where it would not make sense given the context.

In both cases however, the function key press will be queued for execution when possible.

In order to have the basic interpreter not check for function key presses (i.e., to unarm the function key), execute the command

KEY(numeric expression) OFF

Another way is to give a line number of 0 as a GOSUB destination.

Remember, the action of an armed function key is only valid during a running program. If you are in the basic imperative mode or if the program is asking for input from the keyboard, a function key press will produce the default or user-defined string as a response.

EXAMPLE
The following example shows how to use the KEY command to display the time on the bottom left of the screen whenever the F1 key is pressed.

```
100 ON KEY (1) GOSUB 200
110 KEY(1) ON
120 GOTO 120
200 DISPLAY AT(24,1):TIME$
210 RETURN
```

KILL KILL

Format
KILL file-specification

Cross Reference
CLOSE

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

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

The file-specification indicates the name of the file to be deleted. The file-specification is a string-expression; if you use a string constant, you must enclose it in quotation marks.

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

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

Example

KILL "DSK1.MYFILE"
Deletes the file named MYFILE from the diskette in disk drive 1.

Program

The following program illustrates a use of KILL.

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

NOTE: For TI 99/4A PROGRAMS

Delete will no longer be used to delete files from disk storage device (see KILL, CLOSE, FILES). However programs that contain a "DELETE" file statement will execute exactly as they did under TI BASIC or TI EXTENDED BASIC. The token used internally will now be occupied by the KILL command. As long as the program is stored in tokenized form (program file, or DV163 merge format) the execution will not be affected. On listing the program the word "KILL" will be listed instead of "DELETE".