MYARC

T.M.

MYARC
128K.OS
EXTENDED BASIC LEVEL IV

- Performs all of the same features as TI* Extended Basic!
- Execution up to three times faster!
- 40 Character display text mode.
- Vastly improved error handling support!
- Integer Variables! Now you can achieve full support of integer variables.
- Windowing!
- Hi Resolution Graphics! With commands such as DRAW, FILL, CIRCLE, RECT., and many more, you can perform tasks with precision and speed never possible before!
  - CALL MARGIN
  - CALL POINT
  - CALL POINTSTAT
  - CALL DRAWTO
  - CALL WRITE
  - CALL GRAPHICS
  - and many more

*TI is a registered trademark for Texas Instruments, Inc.

[text of front cover]

Myarc™

MYARC
128K.OS
EXTENDED BASIC LEVEL IV

- Performs all of the same features as TI* Extended BASIC!
- Execution up to three times faster!
- 40 Character display text mode
- Vastly improved error handling support
- Integer Variables! Now you can achieve full support of integer variables.
- Windowing!
- Hi Resolution Graphics! With commands such as DRAW, FILL, CIRCLE, RECT, and many more, you can perform tasks with precision and speed never possible before!
  - CALL MARGIN
  - CALL POINT
  - CALL POINT
  - CALL DRAWTO
  - CALL WRITE
  - CALL GRAPHICS
  - and many more

* TI is a registered trademark of Texas Instruments, Inc

IMPORTANT NOTICE REGARDING PROGRAMS AND BOOK MATERIALS

The following should be read and understood before purchasing and/or using MYARC Extended BASIC II.

MYARC does not warrant that the programs contained in the MYARC Extended BASIC II module and accompanying book materials will meet the specific requirements of the consumer, or that the programs and book materials will be free from error. The consumer assumes complete responsibility for any decision made or actions taken based on information obtained using these programs and book materials. Any statements made concerning the utility of MYARC's programs and book material are not to be construed as express or implied warranties.

MYARC MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THESE PROGRAMS OR BOOK MATERIALS OR ANY PROGRAMS DERIVED THEREFROM AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL MYARC BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN PROGRAMS OR BOOR MATERIALS, AND THE SOLE AND EXCLUSIVE LIABILITY OF MYARC, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THIS MODULE. MOREOVER, MYARC SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER AGAINST THE USER OF THESE PROGRAMS OR BOOK MATERIALS BY ANY OTHER PARTY.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you.

COPYRIGHT

DISCLAIMER OF WARRANTY

MYARC, INC. makes no representation of 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 the 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.

TEXAS INSTRUMENTS
HOME COMPUTER

MYARC Extended BASIC II

For The
TI-99/4A Home Computer

IMPORTANT

MYARC Extended BASIC II performs with more power, speed and sophistication than TI Extended BASIC, and therefore requires a larger random access memory (RAM) than does TI Extended BASIC.

A *high-performance* memory expansion with a minimum of 128 Kbytes of RAM such as MYARC's 128, or 512 Kbyte MEXP-1 Memory Expansion must be used with this advanced programming system. MYARC's high-performance Cards* have been especially designed for maximum speed and trouble-free operation with MYARC Extended BASIC II.

PLEASE NOTE

NOT ALL LARGE-CAPACITY MEMORY EXPANSION CARDS
ARE ADEQUATELY DESIGNED TO FULLY FUNCTION
WITH MYARC Extended BASIC II.

* MYARC's 128, and 512 Kbyte MEXP-1 Memory Expansion Cards also provide RAM-disk and Print Spooler functions.

Table of Contents

1. INTRODUCTION

In this manual we assume that you have a working knowledge of TI Extended BASIC and that you are experienced in programming.

Since there are numerous additions to, and improvements over TI Extended BASIC, we recommend that you read this entire manual before using MYARC Extended BASIC II. The many examples, programs, and appendices can be very useful for practice and review.

MYARC's Extended BASIC II is an improved programming language. Its new features add significant capabilities and power to TI Extended BASIC. Program operation and execution is up to four times faster.

MYARC Extended BASIC II improvements include the following:

- INT
  This 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.

- DEFINT
  A new instruction enabling you to declare the data-type of specified numeric variables as DEFINT. DEFINT variables are processed faster and require less memory than do real, or double precision floating point numbers. If DEFINT is not specified, double precision floating point variables will be used. These variables have a greater range of values than DEFINT variables and may contain decimal portions. A numeric variable of the DEFINT data-type is a whole number greater than or equal to -32768 and less than or equal to 32767.

- CALL GRAPHICS
  A subprogram which gives you the option of having either a 32 or 40 column node. With a 32 column mode, there will be 2 columns of margin on each side and 28 columns between. With the 40 column mode, the margins remain the same, but with 36 columns between.

- CALL MARGINS
  This subprogram allows the margins to be removed and screen windows to be specified.

- TERMCHAR
  This function returns the character code of the key pressed to exit from the previously executed statement. If TERMCHAR is used as part of a command, the value returned depends upon which key was pressed to enter the command (**ENTER**, **UP ARROW**, or **DOWN ARROW**).

2. FEATURES OF MYARC Extended BASIC II

2.1. Expanded and Enhanced Graphic Command Set.

MYARC Extended BASIC II dramatically increases the graphics capabilities of your computer. By enhancing already existing Extended BASIC commands and allowing programmer access to standard, text, and bitmap (high-resolution) modes you now have the capabilities of a professional graphics development system.

2.2. New Graphic Modes.

Previously, you could create and run your Extended BASIC programs only in the standard "pattern" mode. This provided you with sprites, color sets, and a 28/32 column by 24 row screen display.

- TEXT MODE
  Text Mode differs from Pattern Mode by giving you a 40 column by 24 row screen display to work with. If you are using a standard TV for display, you will note that all 40 columns are visible. (In the standard mode usually the first and last 1 or 2 columns are lost.)

  Text Mode is most useful in data/word processing, text manipulation, and utility programs not requiring exotic graphics. (Sprites and color sets do not exist in Text Mode.)

- HIGH-RESOLUTION MODE
  High-Resolution Mode (Bitmap Mode) is a high resolution graphics mode which allows placing or locating individual pixels. In contrast, Pattern Mode only allows you to control screen display by placing characters on the screen. If you've used the command "CALL CHAR" to define your own characters before, you know that each character is made up of many dots that are on or off. A standard character is made up of 64 of these dots. In High-Resolution Mode, each of these dots (referred to as pixels), can be directly placed on the screen. Any group of 4 pixels can be made to be any one of the available colors. The screen display is now 256 columns by 192 rows. Add to this the ability to use sprites, and you begin to see the potential.

2.3. Additional Graphic Commands

Additional commands are provided to most effectively utilize these new capabilities. For example, in High-Resolution Mode — CALL DRAW, CALL DRAWTO, CALL FILL, CALL CIRCLE CALL RECTANGLE, CALL POINT, CALL GCHAR (in High-Resolution Mode), and CALL WRITE, along with most of the already existing Extended BASIC commands, provide a medium previously found only in the most complex assembly language programs.

The summary provided below highlights the capabilities of the new graphic commands:

CALL DRAW
Draws, erases or inverts a line from a user defined point to another user-defined point.

CALL DRAWTO
Draws, erases or inverts a line from the last point drawn to a user-defined point.

CALL CIRCLE
Draws, erases or inverts a circle around a specified point with a radius of up to 340 pixels.

CALL POINT
Draws erases or inverts a simple user-defined point (pixel).

CALL RECTANGLE
Draws, erases or inverts squares or rectangles, solid or hollow, from a 1 dot square to a 256 by 192 pixel rectangle.

CALL FILL
Fills in a shape of any dimensions (within the screen boundaries) with a user-defined pattern.

CALL GCHAR
Returns a value indicating whether the point specified was turned off or turned on.

CALL WRITE
Allows text and predefined characters to be placed on the screen at a specified point.

CALL DCOLOR
Defines the current foreground and background colors used in the special High-Resolution Mode graphic commands.

CALL GRAPHICS
Allows access to and from the 3 graphic modes.

2.4. Graphic Enhancements

2.4.1. ENLARGED CHARACTER SET

The predefined and user defined character set has been enlarged from 112 characters (32-143) to 256 characters (0-255).

2.4.2. INCREASED NUMBER OF COLOR SETS.

The number of character color sets has been proportionately increased from 15 sets to match the increased size of the character set. This gives a total of 33 color sets.

2.4.3. NUMBER OF SPRITES

The number of sprites available for use is increased to 32.

2.4.4. SPRITE CONTROL

CALL COINC, CALL POSITION, and CALL DISTANCE all show vastly improved characteristics due to the high speed of MYARC Extended BASIC II. This provides arcade-like animation and response in a BASIC language.

3. SET-UP INSTRUCTIONS

3.1. Memory Expansion

1. Carefully following instructions in your Peripheral Expansion System User Manual, remove the 32K Memory Expansion Card from your PEB if there is one. The 32K Memory Card *MUST NOT* be installed in your PEB when using MYARC Extended BASIC II or when the 128K Memory Expansion Card is in place in your PEB.

2. Insert a high-performance 128K (or larger) Memory Expansion Card into your PEB. You can use the same slot previously occupied by the 32K Card or any other convenient slot.

3.2. Powering Up

Follow standard procedure for powering up peripherals and console and for inserting the MYARC Extended BASIC II module into the module slot of the console.

After each powering up insert the supplied MYARC Extended BASIC II diskette into drive #1 and select "128KOS" from the selection menu on your screen. The MYARC Extended BASIC II program will then load into RAM.

As in TI Extended BASIC, the program will look for DSK1.LOAD and execute accordingly.

If a DSK1.LOAD program is not found, the screen will show

```
MYARC BASIC 2.12
* READY *
```

with a blinking cursor. You may then remove the MYARC Extended BASIC II diskette from drive #1.

PLEASE NOTE — If you have the late model 99/4A (2.2 version) console, the following additional step is required:

After inserting the MYARC Extended BASIC II diskette into drive #1, select TI BASIC from the selection menu of your screen and type in:

```
CALL 128KOS
```

Execution will then proceed as above.

4. REFERENCE SECTION

This reference section is an alphabetical listing of all MYARC Extended BASIC II commands statements, and functions, with detailed explanation on each.

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

Nevertheless, we suggest that you carefully review this entire section.

In addition to newly-added commands, statements and functions, additional details and explanations have been expanded and/or added to the original TI Extended BASIC in the many places where MYARC Extended BASIC II provides additional power, flexibility, and/or sophistication.

4.1. ABS

4.1.1. Format

`ABS(`*`numeric-expression`*`)`

4.1.2. Type

Numeric (REAL or DEFINT)

4.1.3. 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.

4.1.4. 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

4.2. ACCEPT

4.2.1. Format

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

4.2.2. Cross Reference

GRAPHICS, INPUT, LINPUT, MARGINS, TERMCHAR

4.2.3. 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 **AID**, **BACK**, **BEGIN**, **CLEAR**, **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.

In High-Resolution Mode, ACCEPT has no effect. See *Appendix K*.

4.2.4. 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.

4.2.5. 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.

4.2.6. 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.)

4.3. ASC

4.3.1. Format

`ASC(`*`string-expression`*`)`

4.3.2. Cross Reference

CHR$

4.3.3. 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.

4.3.4. 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.

4.4. ATN

4.4.1. Format

`ATN(`*`numeric-expression`*`)`

4.4.2. Cross Reference

COS, SIN, TAN

4.4.3. 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.

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

4.4.4. Examples

```
100 PRINT 4*ATN(-1)
```

Prints -3.141592654.

```
100 Q=PI/ATN(1.732)
```

Sets Q equal to 3.000036389.

4.5. BREAK

4.5.1. Format

`BREAK(`*`line-number-list`*`)`

4.5.2. Cross Reference

CONTINUE, ON BREAK, UNBREAK

4.5.3. 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*.

4.5.4. Breakpoints

When your program stops at a breakpoint, the message `BREAKPOINT IN (LINE NUMBER)` is displayed. While your program is stopped at a breakpoint, you can enter any valid command.

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

- It restores the default character definitions of all characters.
- If the computer is in High-Resolution Mode, it restores the default graphics mode (Pattern) and margin settings (3,30,1,24)
- 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. If the computer is in Pattern or Text Mode, the graphics mode and margin settings remain unchanged.

4.5.5. 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.

4.5.6. 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.

If your program is operating in the High-Resolution Mode, no message is displayed. See *Appendix K*.

4.5.7. 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.

4.6. BYE

4.6.1. Format

BYE

4.6.2. Description

The BYE command resets the computer. Always use BYE to exit from MYARC Extended BASIC II. 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 Extended BASIC II.
- Display the master title screen.

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

4.7. CALL

4.7.1. Format

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

4.7.2. Cross Reference

SUB

4.7.3. 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 Extended BASIC II subprogram or to call a subprogram that you write. After the subprogram is executed, program control returns to the statement immediately following the CALL statement.

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

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

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

4.7.4. 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
```

4.8. CHAR subprogram

4.8.1. Format

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

4.8.2. Cross Reference

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

4.8.3. 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 Mode, you can use the SPRITE subprogram to display sprites on the screen.

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

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

- In High-Resolution Mode, the displayed character remains unchanged.

4.8.4. Pattern and High-Resolution Modes

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

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

In High-Resolution Mode, you can use the HCHAR, VCHAR, and WRITE subprogram to display characters on the screen. Only characters 0-215 are available. See *Appendix K*.

4.8.5. Text Mode

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

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

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

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

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 turned off.

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

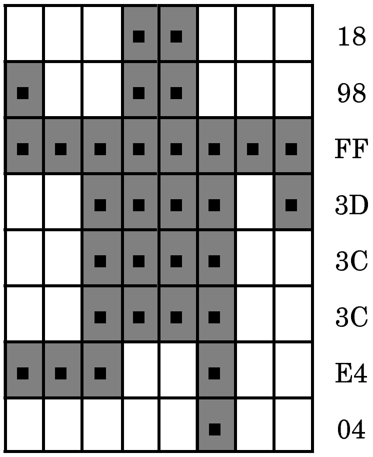
Block				Binary Code (0=off, 1=on)	Hexadecimal Notation
				0000	0
			■	0001	1
		■		0010	2
		■	■	0011	3
	■			0100	4
	■		■	0101	5
	■	■		0110	6
	■	■	■	0111	7
■				1000	8
■			■	1001	9
■		■		1010	A
■		■	■	1011	B
■	■			1100	C
■	■		■	1101	D
■	■	■		1110	E
■	■	■	■	1111	F

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

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

4.8.7. 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 the High-Resolution Mode, make the following changes.

```
105 CALL GRAPHICS(3)
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.

Exiting High-Resolution Mode resets all characters.

The following example works in all graphics modes.

```
100 CALL CLEAR
110 CALL GRAPHICS(X)
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 in line 110 must be replaced with the number of the graphics mode to be designated.

4.9. CHARPAT subprogram

4.9.1. Format

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

4.9.2. Cross Reference

CHAR

4.9.3. 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 the available characters.

4.9.4. Example

```
100 CALL CHARPAT(33,C$)
```

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

4.10. CHARSET subprogram — Set Characters

4.10.1. Format

```
CALL CHARSET
```

4.10.2. Cross Reference

CHAR, COLOR

4.10.3. Description

The CHARSET subprogram restores default character definitions and colors.

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

See *Appendix B* for a list of the available characters.

4.11. CHR$ function — Character

4.11.1. Format

`CHR$(`*`character-code`*`)`

4.11.2. Type

String

4.11.3. Cross Reference

ASC

4.11.4. 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.

4.11.5. Examples

```
100 PRINT CHR$(72)
```

Prints H.

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

Sets X$ equal to 1.

4.11.6. 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
150 GOTO 150
RUN
```

(Press **CLEAR** to stop the program.)

4.12. CIRCLE subprogram

4.12.1. Format

```
CALL CIRCLE(line-type,pixel-row,pixel-column,radius
[,pixel-row2,pixel-column2,radius2[,...]])
```

4.12.2. Cross reference

DCOLOR, DRAW, DRAWTO, FILL, GRAPHICS, POINT, RECTANGLE, WRITE

4.12.3. Description

The CIRCLE subprogram enables you to draw or erase circles around a specified point.

Note that CIRCLE draws and erases the outside edges (perimeter) of a circle, not the area it encircles.

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

TYPE	*ACTION*
2	Reverses the status of each pixel of the circle. (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 circle.
1	Draws a circle of the foreground color specified by the DCOLOR subprogram. This is accomplished by turning on each pixel of the specified circle.
0	Erases a circle. This is accomplished by turning off each pixel of the specified circle.

Pixel-row and *pixel-column* are numeric expressions whose values represent the screen portion of the point the circle is drawn around (center-point). *Radius* is a numeric expression whose value represents the distance from the center-point of the circle to it's outer edge.

You can optionally draw more circles by specifying additional sets of pixels (center-points), and additional radii.

Pixel-row must have a value from 1 to 192. *Pixel-column* must have a value from 1 to 256. *Radius* must have a value from 1 to 320.

The *pixel-row/pixel-column* you specify (center-point), becomes the current position used by the DRAWTO subprogram.

CIRCLE can only be used in High-Resolution Mode. An error results if you use CIRCLE in Pattern or Text Modes.

4.12.4. Example

The following program first selects the High-Resolution Mode (which also clears the screen). Next it uses a FOR-NEXT loop to create and subsequently erase a series of circles from one side of the screen to the other side of the screen, giving the illusion of a rolling ball.

```
100 CALL GRAPHICS(3)
110 FOR T=24 to 216 STEP 4
120 CALL CIRCLE(1,166,28+T,24)
130 CALL CIRCLE(0,166,26+T,24)
140 CALL CIRCLE(1,166,30+T,24)
150 CALL CIRCLE(0,166,28+T,24)
160 NEXT T
170 END
```

4.13. CLEAR subprogram

4.13.1. Format

```
CALL CLEAR
```

4.13.2. Cross Reference

DCOLOR, DELSPRITE

4.13.3. Description

The CLEAR subprogram erases the screen.

In Pattern and Text Modes, CLEAR places a space character (ASCII code 32) in every screen position.

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

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

4.13.4. 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,"0103070FIF3F7FFF")
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(3)
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.)

4.14. CLOSE

4.14.1. Format

```
CLOSE #file-number[:DELETE]
```

4.14.2. Cross Reference

DELETE, OPEN

4.14.3. 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.

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

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

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

4.14.4. 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 LIST command with the file-specification option.
- Enter the BYE, MERGE, NEW, OLD, RUN, or SAVE command.

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

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

4.14.5. Examples

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

4.14.5.1. Cassette File

```
100 OPEN #24:"CS1",INTERNAL,OUTPUT,FIXED
200 CLOSE #24
RUN

REWIND CASSETTE TAPE
THEN PRESS ENTER

PRESS CASSETTE RECORD
THEN PRESS ENTER

PRESS CASSETTE STOP
THEN PRESS ENTER
```

4.14.5.2. 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.

4.15. COINC subprogram — Coincidence

4.15.1. 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`*`)`

4.15.2. Cross Reference

SPRITE

4.15.3. 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.

4.15.4. 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.

4.15.5. 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.

4.15.6. 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.

4.15.7. Program

```
100 CALL CLEAR
110 S$="0103070FlF3F7FFF"
120 CALL CHAR(244,S$)
130 CALL CHAR(250,S$)
140 CALL SPRITE(#1,244,7,50,50)
150 CALL SPRITE(#2,250,5,44,42)
160 CALL COINC(#1,#2,10,C)
170 PRINT C
180 CALL COINC(ALL,C)
190 PRINT C
200 GOTO 200
RUN
–1
0
```

(Press **CLEAR** to stop the program.)

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

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

4.16. COLOR subprogram

4.16.1. Format

Pattern Mode

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

Sprites

CALL COLOR(#*sprite-number*,*foreground-color*[,...l)

4.16.2. Cross Reference

CHAR, DCOLOR, GRAPHICS, SCREEN, SPRITE

4.16.3. 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 Pattern Mode, 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 Extended BASIC II, 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.

See *Appendix K* if you are operating in High-Resolution Mode.

4.16.4. Pattern Mode

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

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.

4.16.5. Text Mode

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

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

4.16.6. High-Resolution Mode

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

4.16.7. 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.

4.16.8. 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 Mode.)

```
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.

4.16.9. 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.)

4.17. CONTINUE

4.17.1. Format

```
CONTINUE
CON
```

4.17.2. Cross Reference

BREAK

4.17.3. 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 **CLEAR (FCTN 4)**. 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 standard characters that have been reset, or any colors. Otherwise, the program continues as if no breakpoint had occurred.

4.18. COS function — Cosine

4.18.1. Format

`COS(`*`numeric-expression`*`)`

4.18.2. Type

REAL

4.18.3. Cross Reference

ATN, SIN, TAN

4.18.4. 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 -1.5707963269514810 or greater than 1.5707963266374210.

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

4.18.5. 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
```

4.19. DATA

4.19.1. Format

```
DATA data-list
```

4.19.2. Cross Reference

READ, RESTORE

4.19.3. 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.

4.19.4. 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.

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

4.20. DCOLOR subprogram— Draw Color

4.20.1. Format

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

4.20.2. Cross Reference

CIRCLE, COLOR, DRAW, DRAWTO, FILL, GRAPHICS, HCHAR, POINT, RECTANGLE, VCHAR WRITE

4.20.3. 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, VCHAR, and WRITE subprogram in High-Resolution Mode.

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

DCOLOR is effective only in High-Resolution Mode. DCOLOR has no effect in Pattern or Text Mode.

4.20.4. 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(3)
120 CALL DCOLOR(5,8)
130 CALL HCHAR(8,20,72,3)
140 GOTO 140
```

(Press **CLEAR** to stop the program.)

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

```
100 CALL CLEAR
110 CALL GRAPHICS(3)
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).

4.21. DEF — Define Function

4.21.1. Format

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

4.21.2. 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 or REAL).

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 or REAL).

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

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

4.21.3. 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.

4.21.4. 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.

4.21.5. 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.)

4.21.6. 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.

4.21.7. Programs

The following program illustrates a use of DEF.

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

In line 100 the parameter B is assigned the DEFINT data-type. In line 110 the value assigned to C is passed to the parameter B.

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

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

4.22. DEFINT

4.22.1. Format

```
DEFINT numeric-variable-list
DEFINT ALL
```

4.22.2. Cross Reference

DEF, DIM, OPTION BASE, REAL, SUB

4.22.3. Description

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

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

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

The *numeric-variable-list* consists of one or more numeric variables separated by commas. The variables are all assigned the DEFINT date-type. A DEFINT statement with a *numeric-variable-list* must have a lower line number than any program reference to any variable in that list.

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

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

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

4.23. DELETE

4.23.1. Format

```
DELETE file-specification
```

4.23.2. Cross Reference

CLOSE

4.23.3. Description

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

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

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.

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

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

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

4.23.4. Example

```
DELETE "DSK1.MYFILE"
```

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

4.23.5. Program

The following program illustrates a use of DELETE.

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

4.24. DELSPRITE subprogram — Delete Sprite

4.24.1. Format

Delete Specified Sprite

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

Delete All Sprites

```
CALL DELSPRITE(ALL)
```

4.24.2. Cross Reference

CLEAR, SPRITE

4.24.3. 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.

4.24.4. 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.

4.24.5. Delete All Sprites

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

4.24.6. 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.

4.25. DIM — Dimension

4.25.1. Format

```
DIM array-name(integer1[,... integer7])[,array-name... ]data-types
```

4.25.2. Cross Reference

DEFINT, OPTION BASE, REAL

4.25.3. 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 REAL array cannot have more than 8191 elements. The number of *integers* in parentheses following the array-name determines the number of dimensions (1-7) in the array.

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

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

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

You cannot use OPTION BASE as a command.

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

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

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

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

4.25.4. 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.

4.25.5. 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.

4.25.6. Examples

```
100 DIM X$(30)
```

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

```
100 DIM D(100),B(l0,9)
```

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

```
100 DEFINT B(10)
```

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

4.26. DISPLAY

4.26.1. Format

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

4.26.2. Cross Reference

DISPLAY USING, GRAPHICS, MARGINS, PRINT

4.26.3. 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 is the PRINT instruction. You can display data at any screen position, sound a tone when data items are displayed, and clear the screen or a portion of the display row before displaying data.

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

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

In High-Resolution Mode, DISPLAY has no effect. *See Appendix K.*

4.26.4. 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.

4.26.5. 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.

4.26.6. 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 ROWl=10*(ROW(1)-48)+ROW(2)-48
250 COLUMNl=10*(COLUMN(1)-48)+COLUMN(2)-48
260 DISPLAY AT(ROW1,COLUMNl)SIZE(1):CHR$(244)
270 GOTO 130
```

(Press **CLEAR** to stop the program.)

4.27. DISPLAY USING

4.27.1. Format

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

4.27.2. Cross Reference

DISPLAY, IMAGE, PRINT

4.27.3. 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 (;).

4.27.4. 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 "##.##".

4.28. DISTANCE subprogram

4.28.1. 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)
```

4.28.2. Cross Reference

COINC, SPRITE

4.28.3. 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.

4.28.4. 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.

4.28.5. A Sprite and a Screen Pixel

The distance between a sprite and 8 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.

4.28.6. 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.

4.29. DRAW subprogram

4.29.1. Format

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

4.29.2. Cross Reference

CIRCLE, DCOLOR, DRAWTO, FILL, GRAPHICS, POINT, RECTANGLE, WRITE

4.29.3. 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 can be used only in High-Resolution Mode. An error results if you use DRAW in Pattern or Text Mode.

In High-Resolution Mode 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.

4.29.4. 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.)

4.30. DRAWTO subprogram

4.30.1. Format

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

4.30.2. Cross Reference

CIRCLE, DCOLOR, DRAW, FILL, GRAPHICS, POINT, RECTANGLE, WRITE

4.30.3. 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 Extended BASIC II, the current position is the intersection of pixel-row 1 and pixel-column 1.

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

DRAWTO can be used only in High-Resolution Mode. An error results if you use DRAWTO in Pattern or Text Mode.

In High-Resolution Mode 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.

4.30.4. 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.)

4.31. END

4.31.1. Format

```
END
```

4.31.2. Cross Reference

STOP

4.31.3. 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.
- If the computer is in High-Resolution Mode, it restores the default graphics mode (Pattern) and margin settings (3, 30, 1, 24).
- 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 reacts 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.

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.

4.32. EOF

4.32.1. Format

`EOF(`*`file-number`*`)`

4.32.2. Type

DEFINT

4.32.3. Cross Reference

ON ERROR

4.32.4. 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.

4.32.5. 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.

4.33. ERR subprogram — Error

4.33.1. Format

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

4.33.2. Cross Reference

ON ERROR

4.33.3. 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-digit 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.

4.33.4. 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.

4.33.5. 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.

4.33.6. Program

The following program illustrates the use of CALL ERR.

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

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

4.34. EXP function — Exponential

4.34.1. Format

`EXP(`*`numeric-expression`*`)`

4.34.2. Type

REAL

4.34.3. Cross Reference

LOG

4.34.4. 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.

4.34.5. 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.

4.35. FILL subprogram

4.35.1. Format

`CALL FILL(`*`pixel-row`*`,`*`pixel-column`*`[,`*`character-code`*`])`

4.35.2. Cross Reference

CIRCLE, DCOLOR, DRAW, DRAWTO, GRAPHICS, POINT, RECTANGLE, WRITE

4.35.3. Description

The FILL subprogram enables you to fill in the area surrounding a specified pixel with a specified pattern and/or 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.

If you do not specify a *character-code*, the default character is a solid square.

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, WRITE subprogram.

FILL can be used only in High-Resolution Mode. An error results if you use FILL in Pattern or Text Mode.

In High-Resolution Mode 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.

4.35.4. 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.)

4.36. FOR TO

4.36.1. Format

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

4.36.2. Cross Reference

NEXT

4.36.3. 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.

4.36.4. 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).

4.36.5. 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.

4.36.6. 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.

4.36.7. 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.

4.36.8. 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.

4.36.9. 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.

4.36.10. 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.)

4.37. FREESPACE function

4.37.1. Format

`FREESPACE(`*`numeric-expression`*`)`

4.37.2. Type

REAL

4.37.3. Description

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

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

4.37.4. 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.

4.37.5. Example

```
PRINT FREESPACE(0)
```

Prints a value that indicates the amount of available memory.

4.38. GCHAR subprogram — Get Character

4.38.1. Format

Pattern and Text Modes

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

High-Resolution Mode

```
CALL GCHAR(pixel-row,pixel-column,numeric-variable)
```

4.38.2. Cross Reference

GRAPHICS, HCHAR, VCHAR

4.38.3. 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.

4.38.4. 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.

4.38.5. 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* must be greater than or equal to 1. In High-Resolution Mode, *pixel-row* must be less than or equal to 192.

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

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

4.38.6. 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.

4.39. GOSUB — Go to a Subroutine

4.39.1. Format

```
GOSUB line-number
GO SUB
```

4.39.2. Cross Reference

ON GOSUB, RETURN

4.39.3. 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 exercise care if you use GOSUB to transfer control to or from a subprogram or into a FOR-NEXT loop.

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

4.39.4. 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 it transfers program control back to the statement immediately following the GOSUB in the first subroutine. Then, when a RETURN is encountered in the first subroutine, program control returns to the statement following the GOSUB in your main program.

4.39.5. 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.

4.39.6. 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
```

4.40. GOTO

4.40.1. Format

```
GOTO line-number
GO TO
```

4.40.2. Cross Reference

ON GOTO

4.40.3. 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.

4.40.4. Program

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

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

4.41. GRAPHICS subprogram

4.41.1. Format

```
CALL GRAPHICS(graphics-mode)
```

4.41.2. Cross Reference

CHAR, CIRCLE, COLOR, DCOLOR, DRAW, DRAWTO, FILL, MARGINS, POINT, RECTANGLE, SCREEN, WRITE

4.41.3. 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 a numeric expression whose value is from 1 to 3, specifying one of the three graphics modes available in MYARC Extended BASIC II.

NUMBER	*MODE*
1	Pattern
2	Text
3	High-Resolution

When you enter MYARC Extended BASIC II, 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 screen margins (3, 30, 1, 24)
- Restores the default current position (pixel-row 1, pixel-column 1).
- Turns off all sound.

4.41.4. 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, RECTANGLE, or WRITE subprogram, the error message `GRAPHICS MODE ERROR IN (LINE NUMBER)` is displayed.

4.41.5. Text Mode

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

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

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

The DCOLOR subprogram has no effect in Text Mode. If you use a CIRCLE, DRAW, DRAWTO, FILL, POINT, RECTANGLE, or WRITE subprogram, the error message `GRAPHICS MODE ERROR IN (LINE NUMBER)` is displayed.

4.41.6. High Resolution Mode

In High-Resolution Mode, you have access to sprites, but not to sprite motion.

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

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

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

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

ACCEPT, DISPLAY, and DISPLAY USING have no effect in High-Resolution Mode. INPUT, LINPUT, PRINT, and PRINT USING are functional only if they are used to access files. Use the command "WRITE" to display messages in High-Resolution Mode. For more information dealing with the restrictions of High-Resolution Mode. see *Appendix R*.

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

4.41.7. A Note on High Resolution Graphics

In High-Resolution Mode 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.

4.41.8. 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.

4.42. HCHAR subprogram — Horizontal Character

4.42.1. Format

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

4.42.2. Cross Reference

DCOLOR, GCHAR, GRAPHICS, VCHAR

4.42.3. 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 l, *row* must be less than or equal to 24. The value of *column* must be greater than or equal to 1. In Pattern or High-Resolution Mode, the *column* must be less than or equal to 32; in Text Mode, *column* must be less than or equal to 40.

HCHAR is not affected by margin settings.

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 High-Resolution Mode the displayed character remains unchanged.

4.42.4. 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.

4.43. IF THEN ELSE

4.43.1. Format

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

4.43.2. 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.

4.43.3. 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, then 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.

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

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

4.43.4. Program

The following program illustrates a use of IF-THEN-ELSE. It accepts up to 1000 numbers ant then prints then 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
```

4.44. IMAGE

4.44.1. Format

`IMAGE` *`format-string`*

4.44.2. Cross Reference

DISPLAY USING, PRINT USING

4.44.3. 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.

4.44.4. 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 to 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 reuses, 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.

4.44.5. Examples

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

IMAGE $####.### allows printing of any number from -999.999 to 9999.999. The following illustrate 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 illustrate 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 ****

4.44.6. Programs

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

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

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

Line 180 prints the values using the IMAGE statements.

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

The amounts are printed with the decimal points aligned.

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

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

4.45. INIT subprogram — Initialize

4.45.1. Format

```
CALL INIT
```

4.45.2. Cross Reference

LINK, LOAD

4.45.3. Description

The INIT subprogram initializes the 8K of memory allocated to assembly-language subprogram.

INIT loads the linkage vectors to support assembly-language subroutines.

INIT removes any assembly-language subprogram that were previously loaded into memory.

INIT reserves almost 8K of memory for assembly-language.

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, a CALL INIT will automatically be performed by the computer.

Although it is not necessary to CALL INIT in your program, you may wish to do so to remove previously loaded subprogram from memory.

4.45.4. Examples

```
CALL INIT
```

Allocates 8K bytes of memory space.

4.46. INPUT

4.46.1. Format

Keyboard Input

`INPUT [`*`input-prompt`*`:]`*`variable-list`*

File Input

`INPUT #`*`file-number`*`[,REC `*`record-number`*`]`

4.46.2. Cross Reference

ACCEPT, EOF, LINPUT, OPEN, REC, TERMCHAR

4.46.3. 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*.

4.46.4. 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 **AID**, **BACK**, **BEGIN**, **CLEAR**, **PROC'D**, **DOWN ARROW**, or **UP ARROW**. You can use the TERMCHAR function to determine which of these keys was pressed to exit From the previous INPUT, LINPUT, or ACCEPT instruction.

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

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

In High-Resolution Mode, keyboard input has no effect. See *Appendix K*.

4.46.5. 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.

4.46.6. 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.)

4.46.7. 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.

4.46.8. 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.

4.46.9. Program

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

```
100 OPEN #1:"CS1",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:"CS1",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

REWIND CASSETTE TAPE
THEN PRESS ENTER

PRESS CASSETTE RECORD
THEN PRESS ENTER

PRESS CASSETTE STOP
THEN PRESS ENTER

REWIND CASSETTE TAPE
THEN PRESS ENTER

PRESS CASSETTE PLAY
THEN PRESS ENTER

PRESS CASSETTE STOP
THEN PRESS ENTER
```

4.47. INT function — Integer

4.47.1. Format

`INT(`*`numeric-expression`*`)`

4.47.2. Type

Real

4.47.3. 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*.

4.47.4. 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.

4.48. JOYST subprogram — Joystick

4.48.1. Format

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

4.48.2. 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 specified joystick is returned in the numeric variables *x* and *y* as follows:

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

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

4.48.3. Example

```
100 CALL JOYST(1,X,Y)
```

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

4.48.4. 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.)

4.49. KEY subprogram

4.49.1. Format

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

4.49.2. Description

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

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

The numeric-expression *key-unit* can have a value from 0 to 5, 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.

4.49.3. 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.
3,4,5	Specific modes for console keyboard.

4.49.4. Status

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

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

0 means no key was pressed.

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

4.49.5. 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.

4.49.6. 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.)

4.50. LEN function — Length

4.50.1. Format

`LEN(`*`string-expression`*`)`

4.50.2. Type

DEFINT

4.50.3. Description

The LEN function returns the number of characters in the string specified by the *string-expression*.

If the *string-expression* is a null string, LEN returns a zero.

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

4.50.4. Examples

```
100 PRINT LEN("ABCDE")
```

Prints 5.

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

Sets X equal to 19.

```
100 DISPLAY LEN("")
```

Displays 0.

```
100 DISPLAY LEN(" ")
```

Displays 1.

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

Displays 5 when A$ equals DAVID.

4.51. LET

4.51.1. Format

`[LET ]`*`variable-list=expression`*

4.51.2. Description

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

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

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

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

The word LET can be optionally omitted from instruction.

4.51.3. Examples

```
100 T=4
```

Assigns to T the value 4.

```
100 X,Y,Z=12.4
```

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

```
100 A=3<5
```

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

```
100 B=12<7
```

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

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

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

4.51.4. Program

The following program illustrates a use of LET.

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

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

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

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

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

4.52. LINK subprogram

4.52.1. Format

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

4.52.2. Cross Reference

INIT, LOAD, SUB

4.52.3. Description

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

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

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

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

4.52.4. Example

```
100 CALL LINK("START",1,3)
```

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

4.53. LINPUT — Line Input

4.53.1. Format

Keyboard Input

`LINPUT [`*`input-prompt`*`:]`*`string-variable`*

File Input

`LINPUT #`*`file-number`*`[,REC `*`record-number`*`]:`*`string-variable`*

4.53.2. Cross Reference

ACCEPT, EOF, INPUT, OPEN, TERMCHAR

4.53.3. Description

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

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

See INPUT for an explanation of keyboard- and file-input, and input options.

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

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

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

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

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

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

4.53.4. Examples

```
100 LINPUT L$
```

Assigns to L$ anything typed before **ENTER** is pressed.

```
100 LINPUT "NAME: "NM$
```

Displays NAME: and assigns to NM$ anything typed before **ENTER** is pressed.

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

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

4.53.5. Program

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

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

4.54. LIST

4.54.1. Format

List to the screen

```
LIST [line-number-range]
```

List to a File (or Device)

```
LIST "file-specification"[:line-number-range]
```

4.54.2. Description

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

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

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

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

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

The LIST command only works with peripherals that support DISPLAY/VARIABLE type records.

4.54.3. The Line-Number-Range

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

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

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

- If *line-number-range* is higher than any line number in the program, the highest-numbered program line is listed.
- If *line-number-range* is lower than any line number in the program, the lowest-numbered program line is listed.
- If *line-number-range* is between lines in the program, the next higher numbered program line is listed.

4.54.4. Examples

```
LIST
```

Lists the entire program in memory on the display screen.

```
LIST 100
```

Lists line 100

```
LIST 100-
```

Lists line 100 and all after it.

```
LIST -200
```

Lists all lines up to and including line 200.

```
LIST 100-200
```

Lists all lines from 100 through 200.

4.55. LOAD subprogram

4.55.1. Format

File Only

```
CALL LOAD(file-specification-list)
```

Data Only

```
CALL LOAD(address,byte-list[,"",address,byte-list[,...]])
```

File and Data

```
CALL LOAD(file-specification-list,address,byte-list[,...])
CALL LOAD(address,byte-list,file-specification-list[,...])
```

4.55.2. Cross Reference

INIT, LINK, PEEK, PEEKV, POKEV, VALHEX

4.55.3. Description

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

To load an assembly-language subprogram, specify a *file-specification-list*; to assign values to CPU memory, specify an *address* and a *byte-list* (an *address* must always be followed by a *byte-list*).

You must enter at least one parameter. The first parameter you specify can be either a *file-specification-list* or an *address*.

If you wish to follow an *address* and *byte-list* with another *address* and *byte-list*, enter a *file-specification-list* or a null string (two adjacent quotation marks) as a separator.

The optional *file-specification-list* consists of one or more file-specifications separated by commas. A file-specification is a string expression; if you use a string constant, you must enclose it in quotation marks.

Each file-specification names an assembly-language object (program) file to be loaded into memory. The specified file can include subprogram names, so that the subprogram can be executed by the LINK subprogram.

The object file to be loaded must be in DISPLAY format with FIXED records (see OPEN). For more information about the file options available with a particular device, refer to the owner's manual that comes with that device.

You can optionally load bytes of data to a specified CPU memory address. The *address* specifies the first address where the data is to be loaded; if the *byte-list* specifies more than one byte of data, the bytes are assigned to sequential memory addresses starting with the address you specify.

The numeric expression *address* must have a value from -32768 to 32767 inclusive.

You can specify an *address* from 0 to 32767 inclusive by specifying the actual address.

You can specify an *address* from 32768 to 65535 inclusive by subtracting 65536 from the actual address. This will result in a value from -32768 to -1 inclusive.

If you know the hexadecimal value of the *address*, you can use the VALHEX function to convert it to a decimal numeric expression, eliminating the possible need for calculations.

If necessary, the *address* is rounded to the nearest integer.

The *byte-list* consists of one or more bytes of data, separated by commas, that are to be loaded into CPU memory starting with the specified *address*.

Each byte in the *byte-list* must be a numeric expression with a value from 0 to 32767. If the value of a byte is greater than 255, it is repeatedly reduced by 256 until it is less than 256. If necessary, a byte is rounded to the nearest integer.

Note that you must use the INIT subprogram to reserve memory space before you use LOAD to load a subprogram.

If you call the LOAD subprogram with invalid parameters or load an object file with absolute (rather than relocatable) addresses, the computer may function erratically or cease to function entirely. If this occurs, turn off the computer, wait several seconds, then turn the computer back on again.

4.55.4. The Loader

LOAD uses a "relocatable linking" loader.

Because it is "relocatable", you cannot use LOAD to specify a memory address at which you want to load a file. However, the file you are loading may specify an absolute load address if it includes an AORG directive.

Because it is "linking", the object files specified in the *file-specification-list* can reference each other.

4.56. LOCATE subprogram

4.56.1. Format

CALL LOCATE(*#sprite-number*,*pixel-row*,*pixel-column*[,...])

4.56.2. Cross Reference

DELSPRITE, SPRITE

4.56.3. Description

The LOCATE subprogram enables you to change the location of one or more sprites.

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

The *pixel-row* and *pixel-column* are numeric expressions whose values specify the screen pixel location of the pixel at the upper-left corner of the sprite.

LOCATE can cause a sprite that has been deleted with DELSPRITE sprite-number to reappear.

4.56.4. Program

The following program illustrates the use of the LOCATE subprogram.

```
100 CALL CLEAR
110 CALL SPRITE(#1,33,7,1,1,25,25)
120 YLOC=INT(RND*150+1)
130 XLOC=INT(RND*200+1)
140 FOR DELAY=1 TO 300:: NEXT DELAY
150 CALL LOCATE(#1,YLOC,XLOC)
160 GOTO 120
```

(Press **CLEAR** to stop the program.)

Line 110 creates a sprite as a fairly quickly moving red exclamation point.

Line 140 locates the sprite at a location randomly chosen in lines 120 and 130.

Line 150 repeats the process.

Also see the third example of the SPRITE subprogram.

4.57. LOG function — Natural Logarithm

4.57.1. Format

`LOG(`*`numeric-expression`*`)`

4.57.2. Type

REAL

4.57.3. Cross Reference

EXP

4.57.4. Description

The LOG function returns the natural logarithm of the value of the *numeric-expression*. LOG is the inverse of the EXP function.

The value of the *numeric-expression* must be greater than zero.

4.57.5. Examples

```
100 PRINT LOG(3.4)
```

Prints the natural logarithm of 3.4, which is 1.223775432.

```
100 X=LOG(EXP(7.2))
```

Sets X equal to the natural logarithm of *e* raised to the 7.2 power, which is 7.2.

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

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

4.57.6. Program

The following program returns the logarithm of any positive number in any base.

```
100 CALL CLEAR
110 INPUT "BASE: ":B
120 IF B=1 THEN 110
130 INPUT "NUMBER: ":N
140 IF N=0 THEN 130
150 LG=LOG(N)/LOG(B)
160 PRINT "LOG BASE";B;"OF";N;"IS";LG
170 PRINT
180 GOTO 110
```

(Press **CLEAR** to stop the program.)

4.58. MAGNIFY subprogram

4.58.1. Format

```
CALL MAGNIFY(numeric-expression)
```

4.58.2. Cross Reference

CHAR, SPRITE

4.58.3. Description

The MAGNIFY subprogram enables you to specify whether all sprites are single or double-sized and whether they are unmagnified or magnified.

The value of the *numeric-expression* specifies the size and magnification "level" of all sprites. (You cannot specify the level of an individual sprite.)

LEVEL	*CHARACTERISTICS*
1	Single-sized, unmagnified
2	Single-sized, magnified
3	Double-sized, unmagnified
4	Double-sized, magnified

The screen position of the pixel in the upper-left corner of a sprite is considered to be the position of that sprite. That pixel remains in the same screen position regardless of changes to the magnification level.

When you enter MYARC Extended BASIC II, sprites are single-sized and unmagnified (level 1). When your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode, the sprite magnification level is restored to 1.

4.58.4. Single-Sized Sprites

A single-sized sprite is defined only by the character you specify when the sprite is created.

4.58.5. Double-Sized Sprites

A double-sized sprite is defined by four consecutive characters, including the character that you specify when the sprite is created.

If the number of the character you specify is a multiple of 4, that character is the first of the four characters that comprise the sprite's definition. If the character number is not a multiple of 4, the next lower character that is a multiple of four is the first character of the sprite.

The first of the four characters defines the upper-left quarter of the sprite, the second character defines the lower-left quarter of the sprite, the third defines the upper-right quarter of the sprite, and the last of the four characters defines the lower-right quarter of the sprite.

4.58.6. Unmagnified Sprites

An unmagnified sprite occupies only the number of characters on the screen specified by the characters that define it.

A single-sized unmagnified sprite occupies 1 character position on the screen; a double-sized unmagnified sprite occupies 4 character positions.

4.58.7. Magnified Sprites

A magnified sprite expands to twice the height and twice the width of an unmagnified sprite. The expansion occurs down and to the right; the pixel in the upper-left corner of the sprite remains in the same screen position.

A magnified sprite has 4 times the area of an unmagnified sprite. When you magnify a sprite, each pixel of the unmagnified sprite expands to 4 pixels of the magnified sprite.

A single-sized magnified sprite occupies 4 character positions on the screen; a double-sized magnified sprite occupies 16 character positions.

4.58.8. Program

The following program illustrates a use of the MAGNIFY subprogram.

A little figure (single-sized, unmagnified) appears near the center of the screen. In a moment, it becomes twice as big (single-sized, magnified), covering four character positions. In another moment, it is replaced by the upper-left corner of a larger figure (single-sized, magnified), still covering four character positions. Then the full figure appears (double-sized, magnified), covering sixteen character positions. Finally it is reduced in size to four character positions (double-sized, unmagnified).

```
100 CALL CLEAR
110 CALL CHAR(148,"1898FF3D3C3CE404")
120 CALL SPRITE(#1,148,5,92,124)
130 GOSUB 230
140 CALL MAGNIFY(2)
150 GOSUB 230
160 CALL CHAR(148,"0103C3417F3F07070707077E7C40000080C0C080
FCFEE2E3E0E0E06060606070")
170 GOSUB 230
180 CALL MAGNIFY(4)
190 GOSUB 230
200 CALL MAGNIFY(3)
210 GOSUB 230
220 STOP
230 REM DELAY
240 FOR DELAY=1 TO 500
250 NEXT DELAY
260 RETURN
```

Line 110 defines character 148.

Line 120 sets up sprite using character 148. By default the magnification factor is 1.

Line 140 changes the magnification factor to 2.

Line 160 redefines character 148. Because the definition is 64 characters long, it also defines characters 149, 150, and 151.

Line 180 changes the magnification factor to 4.

Line 200 changes the magnification factor to 3.

4.59. MARGINS subprogram

4.59.1. Format

```
CALL MARGINS(left,right,top,bottom)
```

4.59.2. Cross Reference

ACCEPT, CLEAR, DISPLAY, DISPLAY USING, GRAPHICS, INPUT, LINPUT, PRINT, PRINT USING, WRITE

4.59.3. Description

The MARGINS subprogram enables you to define screen margins. The margins you specify define a screen window that affects the operation of several instructions.

Left, *right*, *top*, and *bottom* are numeric expressions whose values specify the margins.

The margins cannot "overlap"; that is, the position of the top margin must be higher on the screen than the bottom margin, and the position of the left margin must be farther left on the screen than the right margin.

When creating a screen window, you must leave the window large enough to allow entry of a command.

The valid range for margin location varies according to the graphics mode. Acceptable values for the margins in each mode are as follows:

MODE	*TOP&BOTTOM*	*LEFT&RIGHT*
Pattern	1-24	1-32
Text	1-24	1-40
High-Resolution	1-24	1-32

The upper-left corner of the window defined by the margins is considered to be the intersection of row 1 and column 1 by the ACCEPT, DISPLAY, and DISPLAY USING instructions that use the AT option and the WRITE instruction.

The lower-left corner of the window is considered to be the beginning of the input line by the ACCEPT, INPUT, and LINPUT instructions.

The lower-left corner of the window is considered to be the beginning of the print line by the DISPLAY, DISPLAY USING, PRINT, and PRINT USING instructions.

When the ACCEPT, INPUT, LINPUT, or PRINT USING instructions cause scrolling, scrolling occurs only in the window.

The CLEAR, GCHAR, HCHAR, and VCHAR subprogram are not affected by the margin settings.

In all Modes, the margins can extend to the edges of the screen.

When you enter MYARC Extended BASIC II, the left margin is set to 3 and the right margin to 30. The top and bottom margins are set to 1 and 24 respectively. When a program running in High-Resolution Mode ends, these default margin settings are restored.

4.59.4. Examples

```
100 CALL MARGINS(3,30,1,24)
```

Sets all four margins to the default value in Pattern and High-Resolution Modes.

```
100 CALL MARGINS(1,40,1,24)
```

Sets the left, right, top and bottom margins to the extreme edges of the screen in Text Mode.

4.60. MAX function — Maximum

4.60.1. Format

MAX(*numeric-expression1*,*numeric-expression2*)

4.60.2. Type

Numeric (REAL or DEFINT)

4.60.3. Cross Reference

MIN

4.60.4. Description

The MAX function returns the larger value of two *numeric-expressions*.

MAX is the opposite of the MIN function.

If the values of the *numeric-expressions* are equal, MAX returns that value.

4.60.5. Examples

```
100 PRINT MAX(3,8)
```

Prints 8.

```
100 F=MAX(3E12,1800000)
```

Sets F equal to 3E12.

```
100 G=MAX(-12,-4)
```

Sets G equal to -4.

```
100 A=7::B=-5
110 L=MAX(A,B)
```

Sets L equal to 7 when A=7 and B=-5.

4.61. MERGE

4.61.1. Format

MERGE["]*file-specification*["]

4.61.2. Cross Reference

SAVE

4.61.3. Description

The MERGE command combines a program from an external storage device with the program currently in memory. MERGE is frequently used to combine several previously written program segments into one program.

The *file-specification* is a string constant that indicates the name of the program on the external device. The *file-specification* can optionally be enclosed in quotation marks.

The lines of the external program are inserted in line-number order among the lines of the program in memory. If a line number in the external program duplicates a line number in the program in memory, the new line replaces the old line.

The MERGE command does not clear breakpoints.

A program on an external device can be merged only if it was saved with the MERGE option of the SAVE command.

4.61.4. Example

```
MERGE DSK1.SUB
```

Merges the program SUB into the program currently in memory.

4.61.5. Program

Listed below is an example of how to merge programs. If the following program is saved on DSK1 as BOUNCE with the merge option, it can be merged with other programs.

```
100 CALL CLEAR
110 RANDOMIZE
140 DEF RND50=INT(RND*50-25)
150 GOSUB 10000
10000 FOR AA=1 TO 100
10010 QQ=RND50
10020 LL=RND50
10030 CALL MOTION(#1,QQ,LL)
10040 NEXT AA
10050 RETURN
SAVE "DSK1.BOUNCE",MERGE
NEW
```

Place the following program into the computer's memory.

```
120 CALL CHAR(96,"18183CFFFF3C1818")
130 CALL SPRITE(#1,96,7,92,128)
150 GOSUB 500
160 STOP
```

Now merge BOUNCE with the above program.

```
MERGE DSK1.BOUNCE
```

The program that results from merging BOUNCE with the above program is shown here.

```
LIST
100 CALL CLEAR
110 RANDOMIZE
120 CALL CHAR(96,"18183CFFFF3C1818")
130 CALL SPRITE(#1,96,7,92,128)
140 DEF RND50=INT(RND. 50-25)
150 GOSUB 10000
160 STOP
10000 FOR AA=1 TO 100
10010 QQ=RND50
10020 LL=RND50
10030 CALL MOTION(#1,QQ,LL)
10040 NEXT AA
10050 RETURN
```

Note that line 150 is from the program that was merged (BOUNCE), not from the program that was in memory.

4.62. MIN function — Minimum

4.62.1. Format

`MIN(`*`numeric-expression1`*`,`*`numeric-expression2`*`)`

4.62.2. Type

Numeric (REAL or DEFINT)

4.62.3. Cross Reference

MAX

4.62.4. Description

The MIN function returns the smaller value of two *numeric-expressions*. MIN is the opposite of the MAX function.

If the values of the *numeric-expressions* are equal, MIN returns that value.

4.62.5. Examples

```
100 PRINT MIN(3,8)
```

Prints 3.

```
100 F=MIN(3E12,1800000)
```

Sets F equal to 1800000.

```
100 G=MIN(-12,-4)
```

Sets G equal to -12.

```
100 A=7::B-=5
110 L=MIN(A,B)
```

Sets L equal to -5 when A=7 and B=-5.

4.63. MOTION subprogram

4.63.1. Format

`CALL MOTION(#`*`sprite-number`*`,`*`vertical-velocity`*`,`*`horizontal-velocity`*`[,...])`

4.63.2. Cross Reference

SPRITE

4.63.3. Description

The MOTION subprogram enables you to change the velocity of one or more sprites.

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

The *vertical-velocity* and *horizontal-velocity* are numeric expressions whose values range from -128 to 127. If both values are zero, the sprite is stationary. The speed of a sprite is in direct linear proportion to the absolute value of the specified velocity.

A positive *vertical-velocity* causes the sprite to move toward the bottom of the screen; a negative *vertical-velocity* causes the sprite to move toward the top of the screen.

A positive *horizontal-velocity* causes the sprite to move to the right; a negative *horizontal-velocity* causes the sprite to move to the left.

If neither the *vertical-velocity* nor *horizontal-velocity* are zero, the sprite moves at an angle in a direction and at a speed determined by the velocity values.

When a moving sprite reaches an edge of the screen, it disappears. The sprite reappears in the corresponding position at the opposite edge of the screen.

4.63.4. Program

The following program illustrates a use of the MOTION subprogram.

```
100 CALL CLEAR
110 CALL SPRITE(#1,33,5,92,124)
120 FOR XVEL=-16 TO 16 STEP 2
130 FOR YVEL=-16 TO 16 STEP 2
140 DISPLAY AT(12,11):XVEL;YVEL
150 CALL MOTION(#1,YVEL,XVEL)
160 NEXT YVEL
170 NEXT XVEL
```

Line 110 creates a sprite.

Lines 120 and 130 set values for the motion of the sprite.

Line 140 displays the current values of the motion of the sprite.

Line 150 sets the sprite in motion.

Lines 160 and 170 complete the loops that set the values for the motion of the sprite.

4.64. NEW

4.64.1. Format

NEW

4.64.2. Description

The NEW command erases the program currently in memory, so that you can enter a new program.

The NEW command restores the computer to the condition it was in when you selected MYARC Extended BASIC II from the main selection list, with the following exceptions:

- Memory allocated by the INIT subprogram is not returned to the memory area available to MYARC Extended BASIC II.

- Assembly-language subprogram loaded by the LOAD subprogram remain in memory.

NEW restores all other default values, closes any open files, erases all variable values and names, and cancels any BREAK or TRACE commands in effect.

4.65. NEXT

4.65.1. Format

NEXT *control-variable*

4.65.2. Cross Reference

FOR TO

4.65.3. Description

The NEXT instruction marks the end of a FOR-NEXT loop.

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

The *control-variable* is the same *control-variable* that appears in the corresponding FOR TO instruction.

The NEXT instruction is always paired with a FOR TO instruction to form a FOR-NEXT loop (see FOR TO).

A NEXT statement cannot be part of an IF THEN statement.

IF NEXT is used as a command, it must be part of a multiple-statement line. A FOR TO instruction must precede it on the same line.

4.65.4. Program

The following program illustrates a use of the NEXT statement in lines 130 and 140.

```
100 TOTAL=0
110 FOR COUNT=10 TO 0 STEP -2
120 TOTAL=TOTAL+COUNT
130 NEXT COUNT
140 FOR DELAY=1 TO 100::NEXT DELAY
150 PRINT TOTAL,COUNT;DELAY
RUN
30    -2 101
```

4.66. NUMBER

4.66.1. Format

```
NUMBER [initial-line-number][,increment]
NUM
```

4.66.2. Description

The NUMBER command puts the computer in Number Mode, so that it automatically generates line numbers for your program.

If you enter an *initial-line-number*, the first line number displayed is the one you specify. If you do not specify an *initial-line-number*, the computer starts with line number 100.

Succeeding line numbers are generated by adding the value of the numeric expression *increment* to the previous line number. To specify increment only (without specifying an initial-line-number), you must precede the increment with a comma. The default increment is 10.

If a line number generated by the NUMBER command is the number of a line already in the program in memory, the existing program line is displayed with the line number. To indicate that the displayed line is an existing program line, the prompt symbol (>) that normally appears to the left of the line number is not displayed. When the computer displays an existing program line, you can either edit the line or press **ENTER** to leave the line unchanged.

If you enter a program line that contains an error, the appropriate error message is displayed, and the same line number appears again, enabling you to retype the line correctly.

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

To leave Number Mode, press **CLEAR (FCTN 4)**. If the computer is displaying only a line number (that is, a line number not followed by any characters), you can leave Number Mode by pressing **ENTER**, **UP ARROW**, **DOWN ARROW**, **PROC'D**, **BEGIN**, **AID**, or **BACK**.

4.66.3. Special Editing Keys in Number Mode

In Number Mode, you can use the editing keys whether you are changing existing program lines or entering new ones.

LEFT ARROW (FCTN S) — Pressing **LEFT ARROW** moves the cursor one character position to the left. When the cursor moves over a character, it does not change or delete it.

RIGHT ARROW (FCTN D) — Pressing **RIGHT ARROW** moves the cursor one character position to the right. When the cursor moves over a character, it does not change or delete it.

INS (FCTN 2) — Pressing **INS** enables you to insert characters at the cursor position. Characters that you type are inserted until you press one of the other special editing keys. The character at the cursor position and all characters to the right of the cursor move to the right as you type. You may lose characters if they move so far to the right that they are no longer in the program line.

DEL (FCTN 1) — Pressing **DEL** deletes the character in the cursor position. All characters to the right of the cursor move to the left.

ERASE (FCTN 3) — Pressing **ERASE** erases the program line currently displayed (including the line number). The program line is erected only from the screen, not from memory.

REDO (FCTN 8) — Pressing **REDO** causes the program line or other text moat recently input to be displayed. This line can be especially helpful if you make an error while editing a program line, causing the computer not to accept it. Pressing **REDO** displays the original line so that you can make corrections without having to retype the entire line. When you press **REDO**, the computer leaves Number Mode and enters Edit Mode.

CLEAR (FCTN 4) — Pressing **CLEAR** causes the computer to leave Number Mode. If you were entering a new program line, it is not accepted. If you were changing an existing program line, any changes that you mate are ignored.

ENTER —If you press **ENTER** when the computer is displaying only a line number (that is, a line number not followed by any characters), the computer leaves Number Mode. If the line number is the number of an existing program line, that program line is not changed or deleted.

If you press **ENTER** when the computer is displaying a line number followed by a program line, that line is accepted and the next line number is generated. The displayed line may be a new line that you have entered, an existing program line that you have not changed, or an existing program line that you have edited.

UP ARROW (FCTN E) — **UP ARROW** works exactly the same as **ENTER** in Number Mode.

DOWN ARROW (FCTN X) — **DOWN ARROW** works exactly the same as **ENTER** in Number Mode.

4.66.4. Example

In the following, what you type is UNDERLINED. Press **ENTER** after each line. NUM instructs the computer to number starting at 100 with increments of 10.

```
NUM
100 X=4
110 Z=10
120
NUM 110
110 Z=11
120 PRINT (Y+X)/Z
130
NUM 105,5
105 Y=7
110 Z=11
115
LIST
100 X=4
105 Y=7
110 Z=11
120 PRINT (X+Y)/Z
```

NUM 110 instructs the computer to number starting at 110 with increments of 10. Change line 110 to Z=11.

NUM 105,5 instructs the computer to number starting at line 105 with increments of 5. Line 110 already exists.

4.67. OLD

4.67.1. Format

```
OLD ["]file-specification["]
```

4.67.2. Cross Reference

SAVE

4.67.3. Description

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

The *file-specification* indicates the name of the program to be loaded from the external device. The *file-specification*, a string constant, can optionally be enclosed in quotation marks.

The program to be loaded can be one of the following:

- A saved MYARC Extended BASIC II program.
- A file in DISPLAY/VARIABLE 80 format, created by the LIST command or a text editing or word processing program.
- A specially prepared assembly-language program that execute a automatically when it is loaded.

Before the program is loaded, all open files are closed. The program currently in memory is erased after the program begins to load. For more information see "Loading an Existing Program".

4.67.4. Protected and Unprotected Programs

To execute an unprotected MYARC Extended BASIC II program that has been loaded into memory, enter the RUN command when the cursor appears. You can use the LIST command to display the program or any portion of the program.

If the program was saved using the PROTECTED option of the SAVE command, it starts executing automatically when it is loaded. When the program ends (either normally or because an error) or stops at a breakpoint, it is erased from memory.

4.67.5. Examples

```
OLD CS1
```

Displays instructions and then loads into the computer's memory a program from a cassette recorder.

```
OLD "DSK1.MYPROG"
```

Loads into the computer's memory the program MYPROG from diskette in disk drive one.

```
OLD DSK.DISK3.UPDATE85
```

Loads into the computer's memory the program UPDATE85 from the diskette named DISK3.

4.68. ON BREAK

4.68.1. Format

```
ON BREAK STOP
ON BREAK NEXT
```

4.68.2. Cross Reference

BREAK

4.68.3. Description

The ON BREAK statement enables you to specify the action you want the computer to take when either a breakpoint is encountered or **CLEAR (FCTN 4)** is pressed.

If you enter the STOP option, or if your program does not include an ON BREAK statement, program execution stops when a breakpoint is encountered or **CLEAR** is pressed.

If you enter the NEXT option, program execution continues normally (with the next program statement) when a breakpoint is encountered or **CLEAR** is pressed. If you press **CLEAR** while the computer is performing an input or an output operation with certain external devices, an error condition occurs, causing the program to halt. When the NEXT option is in effect, pressing **QUIT (FCTN =)** is the only way to interrupt your program. However, pressing **QUIT** erases the program in memory and causes you to exit from MYARC Extended BASIC II without closing any open files, possibly causing the loss of data in those files.

ON BREAK does not affect a breakpoint that occurs when a BREAK statement with no line-number-list is encountered in a program.

4.68.4. Program

The following program illustrates the use of ON BREAK.

```
100 CALL CLEAR
110 BREAK 150
120 ON BREAK NEXT
130 BREAK
140 FOR A=1 TO 50
150 PRINT "CLEAR IS DISABLED."
160 NEXT A
170 ON BREAK STOP
180 FOR A=1 TO 50
190 PRINT "NOW IT WORKS."
200 NEXT A
```

Line 110 sets a breakpoint at line 150.

Line 120 sets breakpoint handling to go to the next line.

A breakpoint occurs at line 130 despite line 120, because no line number has been specified after BREAK. Enter CONTINUE.

No breakpoint occurs at line 150 because of line 120; **CLEAR** has no effect during the execution of lines 140 through 160 because of line 120. Line 170 restores the normal use of **CLEAR**.

4.69. ON ERROR

4.69.1. Format

```
ON ERROR STOP
ON ERROR line-number
```

4.69.2. Cross Reference

ERR, GOSUB, RETURN

4.69.3. Description

The ON ERROR statement enables you to specify the action you want the computer to take if a program error occurs.

If you enter the STOP option, or if your program does not include an ON ERROR statement, program execution stops when a program error occurs.

If you enter a *line-number*, a program error causes program control to be transferred to the subroutine that begins at the specified *line-number*. A RETURN statement in the subroutine returns control to a specified program statement.

When an error transfers control to a subroutine, the *line-number* option is canceled. If you wish to restore it, your program must execute an ON ERROR line-number statement again.

The ON ERROR line-number statement does not transfer control when the error is caused by a RUN statement.

4.69.4. Program

The following program illustrates the use of ON ERROR.

```
100 CALL CLEAR
110 DATA "A","4","B","C"
120 ON ERROR 190
130 FOR G=1 TO 4
140 READ X$
150 X=VAL(X$)
160 PRINT X;"SQUARED IS";X*X
170 NEXT G
180 STOP
190 REM ERROR SUBROUTINE
200 ON ERROR 230
210 X$="5"
220 RETURN
230 REM SECOND ERROR
240 CALL ERR(CODE,TYPE,SEVER,LINE)
250 PRINT "ERROR";CODE;" IN LINE";LINE
260 RETURN 170
```

Line 120 causes any error to pass control to line 190.

Line 130 begins a loop. An error occurs in line 150 and control passes to line 190.

Line 200 causes the next error to pass control to line 230.

Line 210 changes the value of X$ to an acceptable value. Line 220 returns control to the line in which the error occurred (line 150).

The second time an error occurs, the SECOND ERROR subroutine is called because of line 200. Line 240 obtains specific information about the error by using CALL ERR. Line 250 reports the nature of the error, and line 260 returns control to line 170 of the main program, which begins the next Iteration of the loop.

When the third error occurs, the message `BAD ARGUMENT IN 150` is displayed because the program does not specify what action to take if another error occurs. Program execution ceases.

4.70. ON GOSUB

4.70.1. Format

```
ON numeric-expression GOSUB line-number-list
GOSUB
```

4.70.2. Cross Reference

GOSUB, RETURN

4.70.3. Description

The ON GOSUB statement enables you to transfer conditional program control to one of several subroutines.

The value of the *numeric-expression* determines to which of the line numbers in the *line-number-list* program control is transferred.

If the value of the *numeric-expression* is 1, program control is transferred to the subroutine that begins at the program statement specified by the first line number in the *line-number-list*; if the value of the *numeric-expression* is 2, program control is transferred to the subroutine that begins at the program statement specified by the second line number in the *line-number-list*; and so on.

If necessary, the value of the *numeric-expression* is rounded to the nearest integer. The value of the *numeric-expression* must be greater than or equal to 1 and less than or equal to the number of line numbers in the *line-number-list*.

The *line-number-list* consists of one or more line numbers separated by commas. Each line number specifies a program statement at which a subroutine begins.

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

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

4.70.4. Examples

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

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

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

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

4.70.5. Program

The following program illustrates a use of ON GOSUB.

```
100 CALL CLEAR
110 DISPLAY AT(11,1):"CHOOSE ONE OF THE FOLLOWING:"
120 DISPLAY AT(13,1):"1 ADD TWO NUMBERS."
130 DISPLAY AT(14,1):"2 MULTIPLY TWO NUMBERS."
140 DISPLAY AT(15,1):"3 SUBTRACT TWO NUMBERS."
150 DISPLAY AT(16,1):"4 EXIT PROGRAM."
160 DISPLAY AT(20,1):"YOUR CHOICE:"
170 DISPLAY AT(22,2):"FIRST NUMBER."
180 DISPLAY AT(23,1):"5ECOND NUMBER."
190 CALL MARGINS(3,30,1,24)
200 ACCEPT AT(20,14)VALIDATE(DIGIT):CHOICE
210 IF CHOICE<1 OR CHOICE>4 THEN 200
220 IF CHOICE=4 THEN STOP
230 ACCEPT AT(22,16)VALIDATE(NUMERIC):FIRST
240 ACCEPT AT(23,16)VALIDATE(NUMERIC):SECOND
250 CALL MARGINS(3,30,1,8)
260 ON CHOICE GOSUB 280,300,320
270 GOTO 190
280 DISPLAY AT(3,1)ERASE ALL:FIRST;"PLUS";SECOND;"EQUALS";FIRST+SECOND
290 RETURN
300 DISPLAY AT(3,1)ERASE ALL:FIRST;"TIMES";SECOND;"EQUALS";FIRST*SECOND
310 RETURN
320 DISPLAY AT(3,1)ERASE ALL:FIRST;"MINUS";SECOND;"EQUALS";FIRST-SECOND
330 RETURN
```

Line 260 determines where to go according to the value of CHOICE.

4.71. ON GOTO

4.71.1. Format

```
ON numeric-expression GOTO line-number-list
                      GOTO
```

4.71.2. Cross Reference

GOTO

4.71.3. Description

The ON GOTO statement enables you to transfer unconditional program control to one of several program statements.

The value of the *numeric-expression* determines to which of the line numbers in the *line-number-list* program control is transferred. If the value of the *numeric-expression* is 1, program control is transferred to the program statement specified by the first line number in the *line-number-list*; if the value of the *numeric-expression* is 2, program control is transferred to the program statement specified by the second line number in the *line-number-list*; and so on.

If necessary, the value of the *numeric-expression* is rounded to the nearest integer. The value of the *numeric-expression* must be greater than or equal to 1 and less than or equal to the number of line numbers in the *line-number-list*.

The *line-number-list* consists of one or more line numbers separated by commas. Each line number specifies a program statement.

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

4.71.4. Examples

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

Transfers control to 1000 if X is 1, 2000 if X is 2, and 300 if X is 3. The equivalent statement using an IF-THEN-ELSE statement is IF X=1 THEN 1000 ELSE IF X=2 THEN 2000 ELSE IF X=3 THEN 300 ELSE PRINT "ERROR!"::STOP.

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

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

4.71.5. Program

The following program illustrates a use of ON GOTO. Line 260 determines where to go according to the value of CHOICE.

```
100 CALL CLEAR
110 DISPLAY AT(11,1):"CHOOSE ONE OF THE FOLLOWING:"
120 DISPLAY AT(13,1):"1 ADD TWO NUMBERS."
130 DISPLAY AT(14,1):"2 MULTIPLY TWO NUMBERS."
140 DISPLAY AT(15,1):"3 SUBTRACT TWO NUMBERS."
150 DISPLAY AT(16,1):"4 EXIT PROGRAM."
160 DISPLAY AT(20,1):"YOUR CHOICE:"
170 DISPLAY AT(22,2):"FIRST NUMBER:"
180 DISPLAY AT(23,1):"SECOND NUMBER:"
190 CALL MARGINS(3,30,1,24)
200 ACCEPT AT(20,14)VALIDATE(DIGIT):CHOICE
210 IF CHOICE<1 OR CHOICE>4 THEN 200
220 IF CHOICE=4 THEN STOP
230 ACCEPT AT(22,16)VALIDATE(NUMERIC):FIRST
240 ACCEPT AT(23,16)VALIDATE(NUMERIC):SECOND
250 CALL MARGINS(3,30,1,8)
260 ON CHOICE GOTO 270,290,310
270 DISPLAY AT(3,1)ERASE ALL:FIRST;"PLUS";SECOND;"EQUALS";FIRST+SECOND
280 GOTO 190
290 DISPLAY AT(3,1)ERASE ALL:FIRST;"TIMES";SECOND;"EQUALS";FIRST*SECOND
300 GOTO 190
310 DISPLAY AT(3,1)ERASE ALL:FIRST;"MINUS";SECOND;"EQUALS";FIRST-SECOND
320 GOTO 190
```

4.72. ON WARNING

4.72.1. Format

```
ON WARNING PRINT
           STOP
           NEXT
```

4.72.2. Description

The ON WARNING statement enables you to specify the action you want the computer to take if a warning condition occurs during the execution of your program.

A warning, a condition caused by invalid input or output, does not normally cause program execution to be terminated.

If you enter the PRINT option, or if your program does not include an ON WARNING statement, the computer displays a warning message when a warning condition occurs during program execution.

If you enter the STOP option, program execution stops when a warning condition occurs during program execution.

If you enter the NEXT option, program execution continues normally when a warning condition occurs and no warning message is displayed. Normally, execution continues beginning with the next program statement; however, if the cause of the warning is an invalid response to an INPUT statement, program execution continues beginning with that same INPUT statement.

You may have multiple ON WARNING statements in the same program.

If your program is running in the High-Resolution Mode, no message is displayed. See *Appendix K*.

4.72.3. Program

The following program illustrates the use of ON WARNING.

```
100 CALL CLEAR
110 ON WARNING NEXT
120 PRINT 120,5/0
130 ON WARNING PRINT
140 PRINT 140,5/0
150 ON WARNING STOP
160 PRINT 160,5/0
170 PRINT 170
RUN
120        9.99999E+**
140
* WARNING
  NUMERIC OVERFLOW IN 140
            9.99999E+**
160
* WARNING
  NUMERIC OVERFLOW IN 160
```

Line 110 sets warning handling to go to the next line. Line 120 therefore prints the result without any message.

Line 130 sets warning handling to the default, printing the message and then continuing execution. Line 140 therefore prints 140, then the warning, and then continues.

Line 150 sets warning handling to print the warning message and then stop execution. Line 160 therefore prints 160 and the warning message and then stops.

4.73. OPEN

4.73.1. Format

```
OPEN #file-number:file-specification[ file-organization[ size]]
[,file-type][,open-mode][,record-type[ record-length]]
```

4.73.2. Cross Reference

CLOSE, INPUT, PRINT

4.73.3. Description

The OPEN instruction establishes an association between the computer and an external device, enabling you to store, retrieve, and process data.

The *file-number* is a numeric expression having a value between 1 and 255. The *file-number* is assigned to the external file or device indicated by the *file-specification*, so that input/output processing instructions may refer to the file by its *file-number*. While a file is open, its *file-number* cannot be assigned to another file. However, you may have more than one file open to a device at one time. *File-number* 0 always refers to the keyboard and screen of your computer, and is always open. You cannot open or close *file-number* 0.

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

The *file-specification* is a string expression; if you use a string constant, you must enclose it in quotation marks.

See *Appendix K* if you are working with files in High-Resolution Mode.

4.73.4. Options

The following options may be entered in any order.

FILE-ORGANIZATION

The *file-organization* specifies whether records are to be accessed sequentially or randomly. Enter SEQUENTIAL for sequential access, or RELATIVE for random access. Records in a sequential-access file are read or written in sequence from beginning to end. Records in a random-access (relative-record) file can be accessed in any order (they can be processed randomly or sequentially).

If you do not specify a file-organization, it is assumed to be SEQUENTIAL.

SIZE

You can optionally specify the initial *size* of the file. *Size* is a numeric expression, the value of which specifies the initial number of records in the file. *Note*: The *size* option cannot be used with all peripherals.

FILE-TYPE

The *file-type* specifies the format of data in the file.

INTERNAL — The computer transfers data in binary format. This is the most efficient method of sending data.

DISPLAY — The computer transfers data in ASCII format. DISPLAY files can only use FIXED records of 64 or 128. If no file-type is specified in OPEN, the default is DISPLAY.

DISPLAY type files require a special kind of output record. Each element in the PRINT field must be separated by a comma enclosed in quotation marks. The comma serves as a field separator in the file. The omission of this comma causes an I/O error. *Note*: This is not the same as a print separator, which must be inserted between an element in the PRINT field and the field separator.

OPEN-MODE

The *open-mode* specifies the input/output operations that can be performed on the file.

INPUT — The computer can only read data from the file.

OUTPUT — The computer can only write data to the file.

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

APPEND — The computer can only write data and only at the end of the file; records already in the file cannot be accessed.

If you open an existing file for OUTPUT, the data items you write to the file replace those currently in the file.

If you do not specify an *open-mode*, it is assumed to be UPDATE.

RECORD-TYPE

The *record-type* specifies whether the records in the file are FIXED (all of the same length) or VARIABLE (of various lengths).

SEQUENTIAL files can have FIXED or VARIABLE records. If you do not specify the *record-type* of a SEQUENTIAL file, it is assured to be VARIABLE.

RELATIVE files must have FIXED records. If you do not specify the *record-type* of a RELATIVE file, it is assumed to be FIXED.

RECORD-LENGTH

You can optionally specify the length of records in the file. *Record-length* is a numeric expression, the value of which specifies the fixed size (for FIXED records) or maximum size (for VARIABLE records) of each record.

If you do not specify a record-length, its value is supplied by the peripheral.

If you open a file that does not exist, a file is created with the options you specify. If you open a file that does exist, the options you specify must be the same as the options that you specified when you created the file, except that a file with FIXED records can be opened as either SEQUENTIAL or RELATIVE, regardless of the *file-organization* that you specified when you created the file.

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

4.73.5. Examples

```
100 OPEN #1:"CS1",OUTPUT,FIXED
```

Opens a file on cassette. The file is SEQUENTIAL, with data stored in DISPLAY format. The file is opened in OUTPUT mode with FIXED length records of 64 bytes.

```
300 OPEN #23:"DSK.MYDISK.X",RELATIVE 100,INTERNAL,FIXED,UPDATE
```

Opens a file named "X". The file is on the diskette named MYDISK in whichever drive that diskette is located. The file is RELATIVE, with data kept in INTERNAL format with FIXED length records of 80 bytes. The file is opened in UPDATE mode and starts with 100 records made available for it.

```
100 OPEN #234:A$,INTERNAL
```

Where A$ equals "DSK2.ABC", assumes a file on the diskette in drive 2 with a name of ABC. The file is SEQUENTIAL, with data kept in INTERNAL format. The file is opened in UPDATE mode with VARIABLE length records that have a maximum length of 80 bytes.

4.73.6. Program

The following program emulates the use of the SIZE option in an OPEN statement.

```
100 OPEN #1:"DSK1.LARGE",RELATIVE
110 PRINT #1,REC 100:0
120 CLOSE #1
130 OPEN #1:"DSK1.LARGE",SEQUENTIAL,FIXED
200 CLOSE #1
```

Line 100 opens a RELATIVE file on diskette.

Line 110 writes to the 100th record, thereby reserving space for 100 contiguous records.

Line 120 closes the file.

Line 130 reopens the file, this time with SEQUENTIAL file organization.

Line 200 closes the file.

4.74. OPTION BASE

4.74.1. Format

```
OPTION BASE 0
            1
```

4.74.2. Cross Reference

DIM, INTEGER, REAL

4.74.3. Description

The OPTION BASE statement enables you to set the lower limit of array subscripts.

You can use the OPTION BASE statement to specify a lower array-subscript limit of either 0 or 1. If your program does not include an OPTION BASE statement, the lower limit is set to 0.

The OPTION BASE statement applies to every array in your program. You can have only one OPTION BASE statement in a program.

If you do not set the lower array-subscript limit to 1, the computer reserves memory for element 0 of each dimension of each array. To avoid reserving unnecessary memory, it is recommended that you set the lower limit to 1 if your program does not use element 0.

The OPTION BASE statement must have a lower line number than any DIM statement or any reference to an array in your program. The OPTION BASE statement is evaluated during pre-scan and is not executed.

The OPTION BASE statement cannot be part of an IF THEN statement.

4.74.4. Example

```
100 OPTION BASE 1
```

Sets the lowest allowable subscript of all arrays to one.

4.75. PATTERN subprogram

4.75.1. Format

```
CALL PATTERN(#sprite-number,character-code[,...])
```

4.75.2. Cross Reference

CHAR, MAGNIFY, SPRITE

4.75.3. Description

The PATTERN subprogram enables you to change the pattern of one or more sprites.

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

Character-code is a numeric expression with a value from 0-255 specifying the character number of the character you want to use as the pattern for a sprite.

If you use the MAGNIFY subprogram to change to double-sized sprites, the sprite definition includes the character specified by the character-code and three additional characters (see MAGNIFY).

4.75.4. Program

The following program illustrates the use of the PATTERN subprogram.

```
100 CALL CLEAR
110 CALL COLOR(12,16,16)
120 FOR A=19 TO 24
130 CALL HCHAR(A,1,120,32)
140 NEXT A
150 A$="01071821214141FFFF414121211907008OE09884848282FFFF8282848498E000"
160 B$="01061820305C4681814246242C180700806018342462428181623A0C0418E000"
170 C$="0106182C2446428181465C3020180700806018040C3A6281814262243418E000"
180 CALL CHAR(244,A$,248,B$,252,C$)
190 CALL SPRITE(#1,244,5,130,1,0,8)
200 CALL MAGNIFY(3)
210 FOR A=244 TO 252 STEP 4
220 CALL PATTERN(#1,A)
230 FOR DELAY=1 TO 15:: NEXT DELAY
240 NEXT A
250 GOTO 210
```

(Press **CLEAR** to stop the program.)

Lines 110 through 140 build a floor.

Lines 150 through 180 define characters 244 through 255.

Line 190 creates a sprite in the shape of a wheel and starts it moving to the right.

Line 200 makes the sprite double-sized.

Lines 210 through 250 make the spokes of the wheel appear to move as the character displayed is changed.

4.76. PEEK subprogram — Peek at CPU RAM

4.76.1. Format

```
CALL PEEK(address,numeric-variable-list
[,"",address,numeric-variable-list[,...]])
```

4.76.2. Cross Reference

LOAD, PEEKV, POKEV, VALHEX

4.76.3. Description

The PEEK subprogram enables you to ascertain the contents of specified CPU memory addresses.

You can use the PEEKV subprogram to ascertain the contents of VDP memory.

The *address* is a numeric expression whose value specifies the first CPU (Central Processing Unit) memory address at which you want to peek.

The *address* must have a value from -32768 to 32767 inclusive.

You can specify an *address* from 0 to 32767 inclusive by specifying the actual address.

You can specify an *address* from 32768 to 65535 inclusive by subtracting 65536 from the actual address. This will result in a value from -32768 to -1 inclusive.

If you know the hexadecimal value of the *address*, you can use the VALHEX function to convert it to a decimal numeric expression, eliminating the need for manual calculations.

If necessary, the *address* is rounded to the nearest integer.

The *numeric-variable-list* consists of one or more numeric-variables separated by commas. Bytes of data starting from the specified CPU memory *address* are assigned sequentially to the numeric-variables in the *numeric-variable-list*.

One byte, with a value from 0 to 255 inclusive, is returned to each specified numeric-variable.

You can specify multiple addresses and numeric-variable-lists by entering a null string (two adjacent quotation marks) as a separator between a *numeric-variable-list* and the next *address*.

If you call the PEEK subprogram with invalid parameters, the computer may function erratically or cease to function entirely. If this occurs, turn off the computer, wait several seconds, and then turn the computer back on again.

4.76.4. Examples

```
100 CALL PEEK(8192,X1,X2,X3,X4)
```

Returns the values in memory locations 8192, 8193, 8194, and 8195 in the variables X1, X2, X3, and X4, respectively.

```
100 CALL PEEK(22433,A,B,C,"",-4276,X,Y,Z)
```

Returns the values in locations 22433, 22434, and 22435 in A, B, C, respectively; and the values in locations 61260, 61261, and 61263 in X, Y, and Z, respectively.

```
100 CALL PEEK(VALHEX("4F55"),V1 V2,V3)
```

Uses VALHEX to ascertain the decimal equivalent of the hexadecimal number 4F55, which is 20309. Then the values in locations 20309, 20310, and 20311 are returned in V1, V2, and V3, respectively.

4.76.5. Program

The following program returns in A the number of the highest numbered sprite (#15) currently in use. A zero is returned to B. because no sprites are defined after the DELSPRITE statement.

```
100 CALL CLEAR
110 CALL SPRITE(#15,33,7,100,100,0,0)
120 CALL PEEK(VALHEX("837A"),A)
130 CALL DELSPRITE(ALL)
140 CALL PEEK(VALHEX("837A"),B)
150 PRINT A,B
```

4.77. PEEKV Subprogram — Peek at VDP RAM

4.77.1. Format

```
CALL PEEKV(address,numeric-variable-list
[,"",address,numeric-variable-list[,...])
```

4.77.2. Cross Reference

LOAD, PEEK, POKEV, VALHEX

4.77.3. Description

The PEEKV subprogram enables you to ascertain the contents of specified VDP memory addresses. You can use the PEEK subprogram to ascertain the contents of CPU memory.

The *address* is a numeric expression whose value specifies the first VDP (Video Display Processor) memory address at which you want to peek.

The *address* must have a value from 0 to 16383 inclusive.

If you know the hexadecimal value of the address (0000-3FFF), you can use the VALHEX function to convert it to a decimal numeric expression.

If necessary, the *address* is rounded to the nearest integer.

The *numeric-variable-list* consists of one or more numeric-variables separated by commas. Bytes of data starting from the specified VDP memory address are assigned sequentially to the numeric variables in the *numeric-variable-list*.

One byte, with a value from 0 to 255 inclusive, is returned to each specified numeric variable.

You can specify multiple addresses and numeric-variable-lists by entering a null string (two adjacent quotation marks) as a separator between a *numeric-variable-list* and the next *address*.

If you call the PEEKV subprogram with invalid parameters, the computer may function erratically. If this occurs, turn off the computer, wait several seconds, then turn the computer back on.

4.77.4. Example

```
100 CALL PEEKV(6300,A1,A2,A3)
```

Returns the values in locations 6300, 6301, and 6302 in A1, A2, and A3, respectively.

4.77.5. Programs

The following program gives an example of the use of PEEKV.

```
100 CALL CLEAR
110 CALL POKEV(32*16+12,66)
120 CALL PEEKV(32*16+12,A)
130 PRINT A
```

Line 110 pokes a "B" into a location that causes it to appear in the middle of the screen. Line 120 peeks at that location, and assigns the value found there (66) to the variable A.

The next program starts a sprite moving diagonally across the screen. Line 120 assigns the values of the row and column coordinates of the sprite to Y and X, respectively.

```
100 CALL CLEAR
110 CALL SPRITE(#1,33,5,100 100,25,25)
120 CALL PEEKV(VALHEX("300"),X,Y)
130 DISPLAY AT(24,1):Y;X
140 GOTO 120
```

(Press **CLEAR** to stop the program.)

4.78. PI function — Pi

4.78.1. Format

PI

4.78.2. Type

REAL

4.78.3. Description

The PI function returns the value of pi.

The value of pi is 3.14159265359.

4.78.4. Example

```
100 VOLUME=4/3*PI*6^3
```

Sets VOLUME equal to four-thirds times pi times six cubed, which is the volume of a sphere with a radius of six.

4.79. POINT subprogram

4.79.1. Format

```
CALL POINT(pixel-type,pixel-row,pixel-column[,pixel-row2,pixel-column2[,...]])
```

4.79.2. Cross Reference

CIRCLE, DCOLOR, DRAW, DRAWTO, FILL, GCHAR, GRAPHICS, RECTANGLE, WRITE

4.79.3. Description

The POINT subprogram enables you to place, or erase specific points (pixels) on the screen, one or more at a time.

Pixel-type is a numeric-expression whose value specifies the action taken by the POINT subprogram.

TYPE	*ACTION*
2	Reverses the status of the specifies point (pixel). (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 pixel.
1	Places a point, of the foreground color specified by the DCOLOR subprogram, at a specified pixel-row and pixel-column. This is accomplished by turning on the pixel at the designated row and column.
0	Erases a point at a specified pixel-row and pixel-column. This is accomplished by turning on the pixel at the designated row and column.

Pixel-row and *pixel-column* are numeric expressions whose values represent the screen position where the point will be placed (turned on or off).

You can optionally place more points by specifying additional sets of pixels.

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

The last *pixel-row*/*pixel-column* you specify becomes the current position used by the DRAWTO subprogram.

POINT can only be used in High-Resolution Mode. An error results if you use POINT in Pattern or Text Modes.

4.79.4. Example

```
100 CALL POINT(1,96,128)
```

Turns on a single pixel in the center of the screen

4.80. POKEV subprogram — Poke to VDP RAM

4.80.1. Format

```
CALL POKEV(address,byte-list[,"",address,byte-list[,...]])
```

4.80.2. Cross Reference

LOAD, PEEK, PEEKV, VALHEX

4.80.3. Description

The POKEV subprogram enables you to assign values directly to specified VDP memory addresses.

You can use the LOAD subprogram to assign values to CPU.

The *address* is a numeric expression whose value specifies the first VDP (Video Display Processor) memory address where data is to be poked. If the *byte-list* specifies more than one byte of data, the bytes are assigned to sequential memory addresses starting with the *address* you specify.

The *address* must have a value from 0 to 16383 inclusive.

If you know the hexadecimal value of the *address* (0000-3FFF), you can use the VALHEX function to convert it to a decimal numeric expression.

If necessary, the *address* is rounded to the nearest integer.

The *byte-list* consists of one or more bytes of data, separated by commas, that are to be poked into VDP memory starting with the specified address.

Each byte in the *byte-list* must be a numeric expression with a value from 0 to 32767. If the value of a byte is greater than 255, it is repeatedly reduced by 256 until it is less than 256. If necessary, a byte is rounded to the nearest integer.

You can specify multiple *addresses* and *byte-lists* by entering a null string (two adjacent quotation marks) as a separator between a *byte-list* and the next *address*.

If you call the POKEV subprogram with invalid parameters the computer may function erratically. If this occurs, turn off the computer, wait several seconds, then turn the computer back on.

4.80.4. Examples

```
100 CALL POKEV(3333,233)
```

Pokes the value 233 into location 3333.

```
100 CALL POKEV(13784,273)
```

Pokes the value 17 (273 reduced by 256 once) into location 13784.

```
100 CALL POKEV(7343,246,"",VALHEX("2E4F"),433)
```

Pokes the value 246 into location 7343, and uses VALHEX to ascertain the decimal equivalent of the hexadecimal number 2E4F (11855). The value 177 (433 reduced by 256 once) is then poked into this location.

4.80.5. Program

The following program uses POKEV to display on the screen the characters that correspond to ASCII codes 65 through 208, at the location specified by the value of R*32+C.

```
100 CALL CLEAR::X=65
110 FOR R=0 TO 23
120 FOR C=0 TO 31 STEP 6
130 CALL POKEV(R*32+C,X)
140 X=X+1
150 NEXT C
160 NEXT R
```

4.81. POS Function — Position

4.81.1. Format

POS(*string-expression*,*substring*,*numeric-expression*)

4.81.2. Type

DEFINT

4.81.3. Description

The POS function returns the position of the first occurrence of a *substring* within a specified string.

The *string-expression* specifies the string within which you are seeking the *substring*. If you use a string constant, it must be enclosed in quotation marks.

The *substring* is the segment (of the *string-expression*) you are trying to locate. The *substring* is a string expression; if you use a string constant, it must be enclosed in quotation marks.

The value of the *numeric-expression* specifies the character position in the *string-expression* where the search for the *substring* begins.

If necessary, the value of the *numeric-expression* is rounded to the nearest integer.

If the *substring* is present within the *string-expression*, POS returns the number of the character position (within the *string-expression*) of the first character of the *substring*.

If the *substring* is not present, or if the value of the *numeric-expression* is greater than the number of characters in the *string-expression*, POS returns a zero.

4.81.4. Examples

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

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

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

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

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

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

```
100 R=POS("PABNAN","AN",l)
```

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

4.81.5. Program

The following program illustrates a use of POS. Input is searched for spaces, and is then printed with each word on a single line.

```
100 CALL CLEAR
110 PRINT "ENTER A SENTENCE."
120 LINPUT X$
130 S=POS(X$," ",1)
140 IF S=0 THEN PRINT X$::PRINT::GOTO 110
150 Y$=SEG$(X$,1,S)::PRINT Y$
160 X$=SEG$(X$,S+1,LEN(X$))
170 GOTO 130
```

(Press **CLEAR** to stop the program.)

4.82. POSITION subprogram

4.82.1. Format

CALL POSITION(#*sprite-number*,*numeric-variable1*,*numeric-variable2*[,...])

4.82.2. Cross Reference

SPRITE

4.82.3. Description

The POSITION subprogram enables you to ascertain the current position of one or more sprites.

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

The current screen position of a sprite is returned as two *numeric-variables* representing the pixel-row and pixel-column, respectively, specifying the position of a screen pixel.

The screen position of the pixel in the upper-left corner of a sprite is considered to be the position of that sprite.

Note that a sprite in motion continues to move during and following the execution of the POSITION subprogram. Remember to allow for this continued motion in your program.

4.82.4. Example

```
100 CALL POSITION(#1,Y,X)
```

Returns the position of the upper left corner of sprite #1. Also see the third example of the SPRITE subprogram.

4.83. PRINT

4.83.1. Format

Print to the Screen

`PRINT [`*`print-list`*`]`

Print to a File (or Device)

`PRINT #`*`file-number`*`[,REC `*`record-number`*`][:`*`print-list`*`]`

4.83.2. Cross Reference

DISPLAY, OPEN, PRINT USING, TAB

4.83.3. Description

The PRINT instruction enables you to display data items on the screen or print them to an external device. You can use PRINT as either a program statement or a command.

The *print-list* consists of one or more print items (items to be printed or displayed) separated by print separators. A PRINT instruction without a *print-list* advances the print position to the first position of the next record. This has the effect of printing a blank record, unless the preceding PRINT instruction ended with a print-separator.

The numeric and/or string expressions in the *print-list* can be constants and/or variables.

Print items are the numeric and string expressions to be printed. Any function is also a valid print item.

Print separators are the punctuation (commas, semicolons, and colons) between print items specifying the placement of the print items in the print record.

4.83.4. Printing to the Screen

Each print item is displayed in the row of the screen window defined by the margins, starting from the far left column of the window. Before a new line is displayed at the bottom of the window, the entire contents of the window (excluding sprites) scroll up one line to make room for the new line. The contents of the top line of the window scroll off the screen and are discarded.

Each line on the screen is treated as one print record. The record length of the screen is the width of the window.

In High-Resolution Mode, attempting to print to the screen has no effect. See *Appendix K*.

4.83.5. Printing to a File

If you include an optional *file-number*, the *print-list* is sent to the specified device. The *file-number* is a numeric expression whose value specifies the number of the file as assigned in its OPEN instruction. You cannot print to a file opened in INPUT mode.

If you do not specify a *file-number* (or if you specify file-number 0), the *print-list* is displayed on the screen.

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

If you print to a file opened in INTERNAL format with FIXED records, each record is filled with trailing binary zeros, if necessary, to bring it to its specified length.

If a record is longer than the record length of the file, it is truncated (extra characters are discarded).

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

4.83.6. Printing Numbers: INTERNAL Files

The amount of memory space allocated to a number printed to a file opened in INTERNAL format varies according to its data-type. A DEFINT is always allocated 3 bytes, whereas a REAL number is always allocated 9 bytes.

Note that if you print a DEFINT value to a file, you cannot access that file on a Home Computer that does not support the INTEGER data-type. You can circumvent this by converting all DEFINT variables and functions to REAL variables before printing them to a file.

4.83.7. Printing Numbers: The Screen and DISPLAY Files

The format of a number printed to the screen or to a file opened in DISPLAY format varies according to the characteristics of the number.

Positive numbers and zero are printed with a leading space (instead of a plus sign); negative numbers are printed with a leading minus sign. All numbers are printed with a trailing space.

Numbers are printed in either decimal form or scientific notation, according to these rules:

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

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

If a number is printed in decimal form, the following rules apply:

- DEFINT numbers and REAL numbers with no decimal portion are printed without decimal points.

- REAL numbers are printed with decimal points in the proper position. if the number has more than 10 digits, it is rounded to 10 digits. A zero is not printed by itself to the left of the decimal point. Trailing zeros after the decimal point are omitted.

If number is printed in scientific notation, the following rules apply:

- The format is *mantissa*E*exponent*.

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

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

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

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

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

4.83.8. Printing Strings

A string constant in a *print-list* must be enclosed in quotation marks. A quotation mark within a string constant is represented by two adjacent quotation marks.

A string printed to a file opened in INTERNAL format has a length one greater than the length of the string.

When a string is printed to the screen or to a file opened in DISPLAY format, no leading or trailing spaces are added to the string.

4.83.9. Print Separators

At least one print separator must be placed between adjacent print items in the *print-list*. Valid print separators are the semicolon (;), the colon and the comma (,):

- A semicolon (;) print separator causes the next print item to print immediately after the current print item.
- A colon (:) print separator causes the next print item to print at the beginning of the next record. Consecutive colons used as print separators must be divided by a space. Otherwise, they are treated as a statement separator symbol.
- If you print to the screen or to a file opened in DISPLAY format, a comma (,) print separator causes the next print item to print at the beginning of the next "zone". Print records are divided into 14-character zones; the number of zones in a print record varies according to its record length. If you print to a file opened in INTERNAL format, a comma print separator has the same effect as a semicolon print separator.

If a print separator would have the effect of splitting the next print item between two records, the print item is moved to the beginning of the following record. However, if discarding the trailing space from a numeric print item allows it to fit in the current record, the number is printed in the current record without its trailing space.

If the *print-list* ends with a print separator, the computer is placed in a print-pending condition. Unless the next PRINT instruction includes the REC option, it is considered to be a continuation of the current PRINT instruction. RESTORE #file-number terminates a print-pending condition.

If the *print-list* is not terminated by a print separator, the computer considers the current record complete when all the print items in the print-list are printed. The first print-item in the next PRINT instruction begins in the next record.

4.83.10. Examples

```
100 PRINT
```

Causes a blank line to appear on the display screen.

```
100 PRINT "THE ANSWER IS";A
```

Causes the string constant THE ANSWER IS to be printed on the display screen, followed immediately by the value of ANSWER. If ANSWER is positive, there will be a blank for the positive sign after IS.

```
100 PRINT X:Y/2
```

Causes the value of X to be printed on a line and the value of Y/2 to be printed on the next line.

```
100 PRINT #12,REC 7:A
```

Causes the value of A to be printed on the eighth record of the file that was opened as number 12 with RELATIVE file organization. (Record number 0 is the fist record.)

```
100 PRINT #32:A,B,C,
```

Causes the values of A, B, and C to be printed on the next record of the file that was opened as number 32. The final comma creates a pending print-condition. The next PRINT statement directed to file number 32 will print on the same record as this PRINT statement unless it specifies a record, or a RESTORE #32 statement is executed, thereby closing the print-pending print condition.

```
100 PRINT #1,REC 3:A,B
150 PRINT #1:C,D
```

Causes A and B to be printed in record 3 of the file that was opened as number 1. PRINT #1:C,D causes C and D to be printed in record 4 of the same file.

4.83.11. Program

The following program prints out values in various positions on the screen.

```
100 CALL CLEAR
110 PRINT 1;2;3;4;5;6;7;8;9
120 PRINT 1,2,3,4,5,6
130 PRINT 1:2:3
140 PRINT
150 PRINT 1;2;3;
160 PRINT 4;5;6/4
RUN
1 2 3 4 5 6 7 8 9
1  2
3  4
5  6
1
2
3

1 2 3 4 5 1.5
```

4.84. PRINT USING

4.84.1. Format

Print to the Screen

```
PRINT USING format-string[:print-list]
            line-number
```

Print to a File (or Device)

```
PRINT #file-number[,REC record-number],USING format-string[print-list]
                                             line-number
```

4.84.2. Cross Reference

IMAGE, PRINT

4.84.3. Description

The PRINT USING instruction enables you to define specific formats for numbers and strings you print.

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

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

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

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

The PRINT USING instruction is identical to the PRINT 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 (;).
- If you use PRINT USING to print to a file, the file must have been opened in DISPLAY format.

4.84.4. Examples

```
100 PRINT USING "###.##":32.5
```

Prints 32.50.

```
100 PRINT USING "THE ANSWER IS ###.#":123.98
```

Prints THE ANSWER IS 124.0.

```
100 PRINT USING 185:37.4,-86.2
185 IMAGE ###.#
```

Prints the values of 37.4 and -86.2 using the IMAGE statement in line 185.

4.85. RANDOMIZE

4.85.1. Format

RANDOMIZE[*seed*]

4.85.2. Cross Reference

RND

4.85.3. Description

The RANDOMIZE instruction varies the sequence of pseudo-random numbers generated by the RND function.

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

The optional *seed* is a numeric expression whose value specifies the random number sequence to be generated by RND functions. The first two bytes of the internal representation of the value of the *seed* determine the random number sequence generated by RND. If the first two bytes of the *seed* are identical each time you run your program, the same random number sequence is generated.

If you do not enter a *seed*, a different and unpredictable sequence of random numbers is generated by RND each time you run your program.

4.85.4. Program

The following program illustrates a use of the RANDOMIZE statement. It accepts a value for the seed and prints the first 10 values obtained using the RND function

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

(Press **CLEAR** to stop the program.)

4.86. READ

4.86.1. Format

`READ` *`variable-list`*

4.86.2. Cross Reference

DATA, RESTORE

4.86.3. Description

The READ statement enables you to assign constants (stored within your program in DATA statements) to variables.

The *variable-list*, consisting of one or more variables separated by commas, specifies the numeric and/or string variables that are to be assigned values. When a READ statement is executed, the variables in its *variable-list* are assigned values from the data-list of a DATA statement. Unless you use a RESTORE statement to specify otherwise, DATA statements are read in ascending line-number order.

If a data-list does not contain enough values to assign to all the variables, the READ statement assigns values from subsequent DATA statements until all the variables have been assigned a value. If there are no more DATA statements, a program error occurs and the message `DATA ERROR IN (LINE NUMBER)` is displayed.

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

See the DATA statement for examples.

4.87. REAL

4.87.1. Format

```
REAL numeric-variable-list
REAL ALL
```

4.87.2. Cross Reference

DEF, DIM, DEFINT, OPTION BASE, SUB

4.87.3. Description

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

REAL variables have a greater range of values than do DEFINT variables and can contain decimal portions. You can use REAL as either a program statement or a command.

The *numeric-variable-list* consists of one or more numeric variables separated by commas. The variables are all assigned the REAL data type. A REAL statement with a *numeric-variable-list* must have a line number lower than any program reference to any variable in that list.

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

When REAL ALL is used as a command, it does not affect any variables unless they follow it on a multiple-statement line.

A REAL ALL statement in your main program does not affect the data type of a numeric variable in a subprogram.

A numeric variable of the REAL data-type can be of any number that can be expressed by the computer.

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

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

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

4.87.4. Examples

```
REAL A
```

As a command, specifies that the variable A is a real number.

```
100 REAL ALL
```

As a statement, specifies that all numeric variables in the program are real numbers, unless specifically declared as DEFINT.

```
100 REAL X(20)
```

Reserves space in memory for 21 real number elements of array X if it is not preceded by an OPTION BASE 1 statement.

4.88. REC function — Record Number

4.88.1. Format

```
REC(file-number)
```

4.88.2. Type

DEFINT

4.88.3. Description

The REC function returns a record number reflecting the position of the next record in the specified file.

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

The REC function returns the number of the record in the specified file that is to be accessed by the next PRINT, INPUT, or LINPUT instruction (the next sequential record). (REC always treats a file as if it were being accessed sequentially, even if it has been opened for relative access.)

The records in a file are numbered sequentially starting with zero.

4.88.4. Example

```
100 PRINT REC(4)
```

Prints the position of the next record in the file that was opened as number 4.

4.88.5. Program

The following program illustrates a use of the REC function.

```
100 CALL CLEAR
110 OPEN #1:"DSK1.PROFILE",RELATIVE,INTERNAL
120 FOR A=0 TO 3
130 PRINT #1:"THIS IS RECORD",A
140 NEXT A
150 RESTORE #1
160 FOR A=0 TO 3
170 PRINT REC(1)
180 INPUT #1:A$,B
190 PRINT A$;B
200 NEXT A
210 CLOSE #1
RUN
0
THIS IS RECORD 0
1
THIS IS RECORD 1
2
THIS IS RECORD 2
3
THIS IS RECORD 3
```

Line 110 opens a file.

Lines 120 through 140 write four records on the file.

Line 150 resets the file to the beginning.

Lines 160 through 200 print the file position and read and print the values at that position.

Line 210 closes the file.

4.89. RECTANGLE subprogram

4.89.1. Format

```
CALL RECTANGLE(line-type,pixel-row1,pixel-column1,
pixel-row2,pixel-column2,pixel-row3,pixel-column3[,...])
```

4.89.2. Cross Reference

CIRCLE, DCOLOR, DRAW, DRAWTO, FILL, GRAPHICS, POINT, WRITE

4.89.3. Description

The RECTANGLE subprogram enables you to place rectangles of various types and proportions on the screen.

Rectangles may be hollow (only the perimeter of the rectangle is drawn), or solid (both the perimeter and the entire area enclosed by the perimeter is drawn).

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

TYPE ACTION

5 Reverses the status of each pixel of the specified rectangle (solid). (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 rectangle.

4 Draws a rectangle (solid), of the foreground color specified by the DCOLOR subprogram. This is accomplished by turning on each pixel in the specified rectangle.

3 Erases a rectangle (solid). This is accomplished by turning off each pixel in the specified rectangle.

2 Reverses the status of each pixel in the perimeter of the specified rectangle. (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 perimeter.

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

0 Erases the perimeter of a rectangle. This is accomplished by turning off each pixel in the specified rectangle.

Pixel-row(#), and *pixel-column*(#), are numeric expressions whose values represent the screen positions of specific points of the rectangle. There are three points needed to define the rectangle, as shown below.

Pixel-row1 / pixel-column1 specify the TOP LEFT corner of the rectangle.

Pixel-row2 / pixel-column2 specify the TOP RIGHT corner of the rectangle.

Pixel-row3 / pixel-column3 specify the BOTTOM LEFT corner of the rectangle.

All *pixel-rows* must have a value from 1 to 192. All *pixel-columns* must have a value from 1 to 256.

Note that the first pixel set (*pixel-row1* and *pixel-column1*) represents the top leftmost point of the rectangle and must have a lower column value than the second pixel set. The second pixel set represents the top rightmost point of the rectangle. In the same manner, the third pixel set, which represents the bottom leftmost point of the rectangle, must have a higher row value than set1 or set2.

If the procedure outlined above is not followed, an error is issued.

You can optionally draw more rectangles by specifying additional sets of pixels. You must specify three sets of pixels for each rectangle.

The bottom-rightmost point of the last rectangle drawn becomes the current position used by the DRAWTO subprogram.

RECTANGLE can only be used in High-Resolution Mode. An error results if you use RECTANGLE in Pattern or Text Modes.

4.89.4. Program

```
100 CALL GRAPHICS(3)
110 CALL RECTANGLE(1,8,80,8,175,134,80)
120 FOR T=1 TO 8:: CALL RECTANGLE(4,T*16,100,T*16,155,T*16+T-1,100):: NEXT T
130 FOR DELAY=1 TO 2000:: NEXT DELAY
140 CALL RECTANGLE(3,16,100,16,155,128,100)
150 FOR DELAY=1 TO 2000:: NEXT DELAY
160 END
```

Line 100 selects High-Resolution Mode (and clears the screen).

Line 110 draws a large box on the screen.

Line 120 uses a FOR-NEXT loop to fill the box with lines of different thickness. (This shows how RECTANGLE could be used to replace DRAW. RECTANGLE is slower, but more versatile).

Line 130 uses a FOR-NEXT loop to delay execution of the next statement.

Line 140 clears the lines, but leaves the box to illustrate how RECTANGLE can be used as an eraser.

Line 150 delays the execution of the next statement.

Line 160 ends the program.

4.90. REM — Remark

4.90.1. Format

```
REM remark
! remark
```

4.90.2. Description

The REM statement enables you to document your program by including explanatory remarks within the program itself.

You can use any character in a remark.

The length of a REM statement is limited only by the length of a program statement.

A REM statement encountered during program execution is ignored by the computer.

4.90.3. Trailing Remarks

In addition to the REM statement, trailing remarks can be added to the ends of lines in MYARC Extended BASIC II, allowing detailed internal documentation of programs. An exclamation mark (!) begins each trailing remark.

4.90.4. Example

```
100 REM BEGIN SUBROUTINE
```

Identifies a section beginning a subroutine.

```
100 FOR X=1 to 16 ! BEGIN LOOP
```

Identifies a section beginning a FOR-NEXT loop.

4.91. RESEQUENCE

4.91.1. Format

```
RESEQUENCE [initial-line-number][,increment]
RES
```

4.91.2. Description

The RESEQUENCE command assigns new line numbers to all lines in the program currently in memory.

If you enter an *initial-line-number*, the first line number assigned is one you specify. If you do not specify an *initial-line-number*, the computer starts with line number 100.

Succeeding line numbers are assigned by adding the value of the numeric expression *increment* to the previous line number. Note that to specify an *increment* only (without specifying an initial-line-number), you must precede the *increment* with a comma. The default *increment* is 10.

To ensure that your program continues to function properly, all line-number references within your program are changed to reflect the newly assigned line numbers. (Line numbers mentioned in REM statements are not affected.) If an invalid line-number reference (a reference to a line number that does not exist in your program) is encountered, tne computer changes the line-number reference to 32767, without displaying any error message or warning.

If the values you enter for the *initial-line-number* and increment would have the effect of creating a line number greater than 32767, the message `BAD LINE NUMBER` is displayed and the program is not resequenced.

4.91.3. Examples

`RES`

Resequences the lines of the program in memory to start with 100 and number by 10s.

`RES 1000`

Resequences the lines of the program to start with 1000 and number by 10s.

`RES 1000,15`

Resequences the lines of the program in memory to start with 1000 and number by 15s.

`RES 15`

Resequences the lines of the program in memory to start with 100 and number by 15s.

4.92. RESTORE

4.92.1. Format

Restore Data

```
RESTORE [line-number]
```

Restore a File

```
RESTORE #file-number[,REC record-number]
```

4.92.2. Cross Reference

DATA, INPUT, PRINT, READ

4.92.3. Description

The RESTORE instruction specifies either the DATA statement to be used with the next READ statement or the record to be accessed by the next file-processing instruction.

4.92.4. RESTORE with DATA and READ Statements

If you enter a *line-number*, the next READ statement executed assigns values beginning from the data-list in the specified DATA statement.

If the specified *line-number* is not the line-number of a DATA statement, the computer uses the first DATA statement with a *line-number* higher than the one you specified.

If there is no higher numbered DATA statement, a program error occurs and the message `DATA ERROR IN (LINE NUMBER)` is displayed (the line-number is the line number of the READ statement that caused the error).

If you do not enter a *line-number* or a *file-number*, the next READ statement executed assigns values beginning from the data-list of the first DATA statement in your program.

If there are no DATA statements in your program, the message `DATA ERROR IN (LINE NUMBER)` is displayed.

4.92.5. RESTORE with a File

If you enter a *file-number*, RESTORE repositions the specified file at its first record, record zero (unless you use the REC option). The *file-number* is a numeric expression whose value specifies the number of the file as assigned in its OPEN instruction.

If you use the REC option, the *record-number* is a numeric expression specifying the number of the record at which you want to position the file. The records in a file are numbered sequentially, starting with zero. The REC option can be used only with a file opened for RELATIVE access.

RESTORE terminates any print- or input-pending conditions.

4.92.6. Examples

```
100 RESTORE
```

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

```
100 RESTORE 130
```

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

```
100 RESTORE #1
```

Sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #1 to be the first record in the file.

```
100 RESTORE #4,REC H5
```

Sets the next record to be used by the next PRINT, INPUT, or LINPUT statement using file #4 to be record H5.

4.93. RETURN

4.93.1. Format

With GOSUB and ON GOSUB

```
RETURN
```

With ON ERROR

```
RETURN [NEXT
        line-number]
```

4.93.2. Cross Reference

GOSUB, ON GOSUB, ON ERROR

4.93.3. Description

The RETURN statement causes program control to return to the main program from a subroutine called by a GOSUB, ON GOSUB, or ON ERROR statement.

4.93.4. RETURN with GOSUB and ON GOSUB

When the computer encounters a RETURN statement in a subroutine called by a GOSUB or ON GOSUB statement, program control returns to the statement immediately following the GOSUB or ON GOSUB statement.

No options are allowed with a RETURN statement in a subroutine called by a GOSUB or ON GOSUB statement.

4.93.5. RETURN with ON ERROR

The action taken by the computer when it encounters a RETURN statement in a subroutine called by an ON ERROR statement depends on the RETURN option.

If you specify the NEXT option, program control returns to the statement immediately following the statement that caused the error.

If you specify a *line-number*, program control is transferred to the specified program statement.

If you do not specify an option, program control returns to the statement that caused the error. The statement is re-executed.

RETURN "clears" the error, so that it can no longer be analyzed by the ERR subprogram.

4.93.6. Programs

The following program illustrates a use of RETURN as used with GOSUB. The program figures interest on an amount of money put into savings.

```
100 CALL CLEAR
110 INPUT "AMOUNT DEPOSITED: ":AMOUNT
120 INPUT "ANNUAL INTEREST RATE: ":RATE
130 IF RATE>1 THEN RATE=RATE*100
140 PRINT "NUMBER OF TIMES COMPOUNDED"
150 INPUT "ANNUALLY: ":COMP
160 INPUT "STARTING YEAR: ":Y
170 INPUT "NUMBER OF YEARS: ":N
180 CALL CLEAR
190 FOR A=Y TO Y+N
200 GOSUB 240
210 PRINT A,INT(AMOUNT* 100+.5)/100
220 NEXT A
230 STOP
240 FOR B=1 TO COMP
250 AMOUNT=AMOUNT+AMOUNT*RATE/(COMP*100)
260 NEXT B
270 RETURN
```

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

```
100 CALL CLEAR
110 A=1
120 ON ERROR 160
130 X=VAL("D")
140 PRINT 140
150 STOP
160 REM ERROR HANDLING
170 IF A>4 THEN 220
180 A=A+1
190 PRINT 190
200 ON ERROR 160
210 RETURN
220 PRINT 220 :: RETURN NEXT
RUN

190
190
190
190
220
140
```

Line 120 causes an error to transfer control to line 160. Line 130 causes an error.

Line 170 checks to see if the error has occurred four times and transfers control to 220 if it has. Line 180 increments the error counter by one. Line 190 prints 190. Line 200 resets the error handling to transfer to line 160. Line 210 returns to the line that caused the error and executes it again.

Line 220, which is executed only after the error has occurred four times, prints 220 and returns to the line following the line that caused the error.

Line 140, the next one after the one that causes the error, prints 140.

Also see example of the ON ERROR statement.

4.94. RND function — Random Number

4.94.1. Format

```
RND
```

4.94.2. Type

REAL

4.94.3. Cross Reference

RANDOMIZE

4.94.4. Description

The RND function returns a pseudo-random number.

RND returns the next pseudo-random number in the current series of pseudo-random numbers. The number returned is always greater than or equal to 0 and less than 1.

The numbers returned by RND are called "pseudo-random" because they are not generated strictly at random, but are generated as members of predefined series. You can use the RANDOMIZE instruction to make the numbers generated by RND more random.

The same sequence of random numbers is generated by RND each time you run a particular program unless the program includes a RANDOMIZE instruction.

4.94.5. Examples

```
100 COLOR16=INT(RND*16)+1
```

Sets COLOR16 equal to some number from 1 through 16.

```
100 VALUE=INT(RND*16)+10
```

Sets VALUE equal to some number from 10 through 25.

```
100 LL(8)=INT(RND*(B-A+1))+A
```

Sets LL(8) equal to some number from A through B.

4.95. RPT$ function — Repeat String

4.95.1. Format

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

4.95.2. Type

String

4.95.3. Description

The RPT$ function returns a string consisting of a specified string repeated a specified number of times.

The *string-expression* specifies the string to be repeated. If you use a string constant, it must be enclosed in quotation marks.

The value of the *numeric-expression* specifies the number of repetitions of the *string-expression*.

If the length of the *string-expression* and the value of the *numeric-expression* would create a string longer than 255 characters, the excess characters are discarded and the following message is displayed;

```
*WARNING
 STRING TRUNCATED
```

4.95.4. Examples

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

Sets M$ equal to "ABCDABCDABCDABCD"

```
100 CALL CHAR(244,RPT$("0000FFFF",8))
```

Defines characters 244 through 247 with the string
"0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF"

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

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

4.96. RUN

4.96.1. Format

Execute Program in Memory

```
RUN [line-number]
```

Execute Program on External Device

```
RUN file-specification
```

4.96.2. Description

The RUN instruction causes the computer either to execute the program currently in memory or to both load and execute a program from an external. You can use RUN as either a program statement or a command.

When you use RUN as a program statement, one program can start the execution of another program. This enables you to divide a large program into smaller segments, each of which can be loaded into memory only as needed.

If you specify a *line-number*, your program starts running at the specified program line.

If you enter a *file-specification*, your program is first loaded into memory from the specified external device, and then executed starting from the lowest-numbered line in the program. The *file-specification* is a string expression; if you use a string constant, you must enclose it in quotation marks.

If you do not enter either a *line-number* or a *file-specification*, the computer executes the program currently in memory starting with the lowest-numbered line in the program.

Before the program starts running, the computer:

- Sets the values of all numeric variables to zero.
- Sets the values of all string variables to null strings (strings containing no characters).
- Closes all open files.
- Restores the default screen color (cyan).
- Deletes all sprites.
- Resets the sprite magnification level to 1.
- Checks for certain program errors.

RUN does not affect the graphics mode, margin settings, graphics colors (see DCOLOR), or current position (see DRAWTO).

4.96.3. Examples

```
RUN
```

Causes the computer to begin execution of the program in memory.

```
RUN 200
100 RUN 200
```

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

```
RUN "DSK1.PRG3"
100 RUN "DSKI.PRG3"
```

Causes the computer to load and begin execution of the program named PRG3 from the diskette in disk drive 1.

```
100 A$="DSK1.MYFILE"
110 RUN A$
```

Causes the computer to load and begin execution of the program named MYFILE from the diskette in disk drive 1.

4.96.4. Program

The following program illustrates the use of the RUN command used as a statement. It creates a "menu" and lets the person using the program choose what other program he wishes to run. The other programs should RUN this program rather than ending in the usual way, so that the menu is given again after they are finished.

```
100 CALL CLEAR
110 PRINT "1 PROGRAM 1."
120 PRINT "2 PROGRAM 2."
130 PRINT "3 PROGRAM 3."
140 PRINT "4 END."
150 PRINT
160 INPUT "YOUR CHOICE: "C
170 IF C=1 THEN RUN "DSK1.PRG1"
180 IF C=2 THEN RUN "DSK1.PRG2"
190 IF C=3 THEN RUN "DSK1.PRG3"
200 IF C=4 THEN STOP
210 GOTO 100
```

4.97. SAVE

4.97.1. Format

```
SAVE file-specification[,MERGE
                         PROTECTED]
```

4.97.2. Cross Reference

MERGE, OLD

4.97.3. Description

The SAVE command copies the program in memory to an external storage device. When you are using SAVE, your program remains in memory, even if an error occurs.

The saved program can later be loaded back into memory with the OLD command.

The *file-specification* names the program to be stored (see "File Specifications). The *file-specification*, a string constant, optionally can be enclosed in quotation marks.

To specify that your program is to be available for merging with other programs, use the MERGE option. If you use the MERGE option, the program is stored as a SEQUENTIAL file in DISPLAY format with VARIABLE records (see OPEN); MERGE can be used only with devices that accept these options.

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

If you do not use the MERGE option, your program cannot later be merged with another program.

If you use the PROTECTED option, you ensure that the program, when subsequently loaded with the OLD command, cannot be listed, edited, or saved.

A protected program starts executing automatically when it is loaded; when the program ends (either normally or because of an error) or stops at a breakpoint, it is erased from memory. As the PROTECTED option is not reversible, it is recommended that you keep an unprotected version of the program. If you also wish to protect a diskette-based program from being deleted, use the protect feature of the Disk Manager cartridge.

SAVE removes any breakpoints you have set in your program.

4.97.4. Examples

```
SAVE DSK1.PRG1
```

Saves the program in memory on the diskette in disk drive 1 under the name PRG1.

```
SAVE DSK1.PRG1,PROTECTED
```

Saves the program in memory on the diskette in disk drive 1 under the name PRG1. The program may be loaded into memory, but it may not be edited, listed, or resaved.

```
SAVE DSK1.PRG1,MERGE
```

Saves the program in memory on the diskette in disk drive 1 under the name PRG1. The program may later be merged with a program in memory by using the MERGE command.

4.98. SAY subprogram

4.98.1. Format

```
CALL SAY(word-string[,direct-string][,... ])
```

4.98.2. Cross Reference

SPGET

4.98.3. Description

The SAY subprogram enables you to instruct the computer to produce speech.

Word-string is a string-expression whose value is any of the words or phrases in the computer's resident vocabulary. If you use a string constant, you must enclose it in quotation marks. Alphabetic characters must be upper-case.

The computer substitutes "UHOH" for a *word-string* not in the vocabulary.

A speech phrase (more than one word) must be enclosed in pound signs (#). A speech phrase must be predefined; that is, it must be resident in the computer's vocabulary.

A compound is a new word formed by combining two words already in the vocabulary. For example, SOME+THING produces "something" and THERE+FOUR produces "therefore". A compound must not be enclosed in pound signs.

See *Appendix H* for a list of the computer's resident vocabulary.

Direct-string is a string expression whose value is the computer's internal representation of a word or phrase. You can use or modify a *direct-string* returned by the SPGET subprogram.

See *Appendix I* for information on adding suffixes to *direct-strings*. You can specify multiple *word-strings* and *direct-strings* by alternating them. To specify two consecutive *word-strings* or *direct-strings*, enter an extra comma as a separator between them.

4.98.4. Examples

```
100 CALL SAY("HELLO, HOW ARE YOU")
```

Causes the computer to say "Hello, how are you".

```
CALL SAY(,A$,,B$)
```

Causes the computer to say the words indicated by A$ and B$, which must have been returned by SPGET.

4.98.5. Program

The following program illustrates using CALL SAY with a word-string and three direct-strings.

```
100 CALL SPGET("HOW",X$)
110 CALL SPGET("ARE",Y$)
120 CALL SPGET("YOU",Z$)
130 CALL SAY("HELLO",X$,,Y$,,Z$)
```

4.99. SCREEN Subprogram

4.99.1. Format

`CALL SCREEN([`*`foreground-color,`*`]`*`background-color`*`)`

4.99.2. Cross Reference

COLOR, DCOLOR, GRAPHICS

4.99.3. Description

The SCREEN subprogram enables you to change the screen color. The screen color is the color of the border and the color displayed when transparent is specified as the *foreground-color* or *background-color* of a character or pixel.

In Text Mode, SCREEN enables you to change the color of the displayed characters, as well as the color of the screen.

Background-color is a numeric expression whose value specifies a screen color from among the 16 available colors.

In Text Mode, *foreground-color* is a numeric expression whose value specifies a color from among the 16 available colors, representing the *foreground-color* of all 256 characters.

If you specify a *foreground-color* and the computer is not in Text Mode, it has no effect. If the computer is in Text Mode and you do not specify *foreground-color*, the *foreground-color* remains unchanged.

When you enter MYARC Extended BASIC II, the *background-color* is cyan and the *foreground-color* is black. When your program ends (either normally or because of an error), stops at a breakpoint, or changes graphics mode, the default colors are restored.

The codes for the available colors are listed in *Appendix F*.

4.99.4. Examples

```
100 CALL SCREEN(8)
```

Changes the screen to cyan, which is the standard screen color.

```
100 CALL SCREEN(2)
```

Changes the screen to black.

4.99.5. Program

The following program uses CALL SCREEN with CALL VCHAR and PRINT in the Text Mode to change the color of a character.

```
100 CALL CLEAR
110 CALL GRAPHICS(2)
120 CALL VCHAR(12,12,33,3)
130 CALL SCREEN(5,16)
140 PRINT "DARK BLUE SCREEN WITH WHITE LETTERS"
150 GOTO 150
```

(Press **CLEAR** to stop the program.)

Line 130 changes the screen to dark blue and the characters to white.

4.100. SEG$ function — String Segment

4.100.1. Format

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

4.100.2. Type

String

4.100.3. Purpose

The SEG$ function returns a specified substring (segment of a string).

The *string-expression* specifies the string of which you want to specify a substring. If you use a string constant, it must be enclosed in quotation marks.

The *start-position* is a numeric expression whose value specifies the character position in the *string-expression* where the substring begins. The value of the *start-position* must be greater than zero.

The *length* is a numeric expression whose value specifies the length of the substring.

If the *start-position* is greater than the *length* of the *string-expression*, or if the length is zero, SEG$ returns a null string.

If the specified *length* is greater than the remaining length of the string-expression (starting from the specified *start-position*), SEG$ returns a substring consisting of all characters in the *string-expression* starting from the *start-position* to the end of the *string-expression*.

4.100.4. Examples

```
100 X$=SEG$("FIRSTNAME LASTNAME",1,9)
```

Sets X$ equal to FIRSTNAME.

```
100 Y$=SEG$("FIRSTNAME LASTNAME",11,8)
```

Sets Y$ equal to LASTNAME.

```
100 Z$=SEG$("FIRSTNAME LASTNAME",10,1)
```

Sets Z$ equal to " ".

```
100 PRINT SEG$(A$,B,C)
```

Prints the substring of A$ starting at the character at position B and extending for C characters.

4.101. SGN function — Signum (Sign)

4.101.1. Format

`SGN(`*`numeric-expression`*`)`

4.101.2. Type

DEFINT

4.101.3. Description

The SGN function returns a number indicating the algebraic sign of the value of the *numeric-expression*.

If the value of the *numeric-expression* is negative, SGN returns a -1.

If the value of the *numeric-expression* is zero, SGN returns a 0.

If the value of the *numeric-expression* is positive, SGN returns a +1.

4.101.4. Examples

```
100 IF SGN(X2)=1 THEN 300 ELSE 400
```

Transfers control to line 300 if X2 is positive and to line 400 if X2 is zero or negative.

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

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

4.102. SIN function — Sine

4.102.1. Format

SIN(*numeric-expression*)

4.102.2. Type

REAL

4.102.3. Cross Reference

ATN, COS, TAN

4.102.4. Description

The SIN function returns the sine of the angle whose measurement in radians is the value of the *numeric-expression*.

The value of the numeric-expression cannot be less than -1.5707963267944E10 or greater than 1.5707963267944E10.

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

4.102.5. Program

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

```
100 A=.5235987755982
110 B=30
120 C=45*PI/180
130 PRINT SIN(A);SIN(B)
140 PRINT SIN(B*PI/180)
150 PRINT SIN(C)
RUN
.5 -.9880316241
.5
.7071067812
```

4.103. SOUND subprogram

4.103.1. Format

```
CALL SOUND(duration,frequency1,volume1[,frequency2,volume2]
[,frequency3,volume3][,frequency4,volume4])
```

4.103.2. Description

The SOUND subprogram enables you to instruct the computer to produce musical tones or noise.

The computer contains three music generators and one noise generator, enabling you to create up to four different sounds at once. You can specify the frequency and volume of each sound independently.

Duration is a numeric expression whose absolute value specifies the length of the sound in milliseconds (thousandths of seconds). *Duration* can have an absolute value from 1 to 4250. (A value of 1000 will produce a sound for one second.)

The actual duration produced by the computer may vary by as much as one sixtieth (1/60) of a second from the value you specify.

You can enter only one *duration*, which applies to all specified sounds (music and noise).

Frequency is a numeric expression that has different meanings depending on whether you use it to specify one of the music generators or the noise generator.

You must enter at least one *frequency*.

The *frequency* of a music generator specifies the frequency of the tone in Hertz (cycles per second). The acceptable values range from 110 to 44733; the upper limit exceeds the range of human hearing.

The actual frequency produced by the computer may vary by as much as ten percent from the value you specify.

See *Appendix C* for the frequencies of some commonly used tones.

The *frequency* of the noise generator has a value from -1 to -8, specifying the type of noise produced.

The *frequencies* from -1 to -3 produce different types of periodic noise. A *frequency* of -4 produces a periodic noise that varies depending on the *frequency* value of the third music generator.

The *frequencies* from -5 to -7 produce different types of white noise. A *frequency* of -8 produces a white noise that varies depending on the *frequency* value of the third music generator.

Volume is a numeric expression whose value is inversely proportional to the loudness of the sound.

You must enter at least one *volume*.

The *volume* can be from 0 to 30. Zero is the maximum volume and 30 is silence.

If you call SOUND while the computer is still producing the tones specified in a previous call to the SOUND subprogram, the result depends on the algebraic sign of the *duration* of the previous call to SOUND. If the *duration* was positive, the new sound does not begin until the old sound is complete. If the *duration* was negative, the new sound begins immediately, interrupting the old sound.

4.103.3. Examples

```
100 CALL SOUND(1000,110,0)
```
Plays A below low C loudly for one second.

```
100 CALL SOUND(500,110,0,131,0,196,3)
```
Plays A below low C and low C loudly, and G below C not as loudly, all for half a second.

```
100 CALL SOUND(4250,-8,0)
```
Plays loud white noise for 4.250 seconds.

```
100 CALL SOUND(DUR,TONE,VOL)
```
Plays the tone indicated by TONE for a duration indicated by DUR, at a volume indicated by VOL.

4.103.4. Program

The following program plays the 13 notes of the first octave that is available on the computer.

```
100 X=2^(1/12)
110 FOR A=1 TO 13
120 CALL SOUND(100,110*X^A,0)
130 NEXT A
```

4.104. SPGET subprogram — Get Speech

4.104.1. Format

CALL SPGET(*word-string*,*string-variable*[,...])

4.104.2. Cross Reference

SAY

4.104.3. Description

The SPGET subprogram enables you to assign the computer's internal representation of a speech word to a variable.

SPGET is especially useful if you want to add a suffix to a word in the computer's resident vocabulary.

Word-string is a string expression whose value is any of the words or phrases in the computer's resident vocabulary. If you use a string constant, you must enclose it in quotes.

The computer substitutes "UHOH" for a *word-string* not in the vocabulary.

A speech phrase (more than one word) must be enclosed in pound signs (#).

See *Appendix H* for a list of the computer's resident vocabulary.

The internal representation of the *word-string* (the direct-string) is returned in the *string-variable*. See *Appendix I* for information on adding suffixes to direct-strings.

You can specify multiple *word-strings* and direct-strings by alternating them.

4.104.4. Program

The following program illustrates using CALL SPGET.

```
100 CALL SPGET("TEXAS INSTRUMENTS",X$)
110 CALL SPGET("COMPUTER",Y$)
120 CALL SAY("I AM A",X$,,Y$)
```

4.105. SPRITE subprogram

4.105.1. Format

CALL SPRITE(#*sprite-number*,*character-code*,*foreground-color*, *pixel-row*,*pixel-column*[,*vertical-velocity*,*horizontal-velocity*][,...])

4.105.2. Cross Reference

CHAR, COINC, COLOR, DELSPRITE, DISTANCE, GRAPHICS, LOCATE, MAGNIFY, MOTION, PATTERN, POSITION, SCREEN

4.105.3. Description

The SPRITE subprogram enables you to create sprites.

Sprites are graphics that can be assigned any valid color and placed anywhere on the screen. Sprites treat the screen as a grid 256 pixels high and 256 pixels wide. However, only the first 192 pixels are visible on the screen.

You can create up to 32 sprites in all modes except Text Mode, which does not allow sprites (the SPRITE subprogram has no effect in Text Mode).

In Pattern Mode, sprites can be set in motion in any direction at a variety of speeds. A sprite continues its motion until it is specifically changed by the program or until program execution stops. Because sprites move from pixel to pixel, their motion can be smoother than that of characters, which can be moved only one character position (8 pixels) at a time.

In High-Resolution Mode, sprites cannot be set in motion. If you specify a vertical or horizontal velocity, it will be ignored.

Sprites "pass over" characters on the screen. When two or more sprites are coincident (occupying the same screen pixel position), the sprite with the lowest sprite-number covers the other sprite(s).

At any given time, only four sprites can be on the same horizontal pixel-row. When five or more sprites are on the same pixel-row, that row of pixels in the sprite(s) with the highest sprite-number(s) disappears.

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

4.105.4. Sprite Specifications

The *sprite-number* is a numeric expression with a value from 1 to 32. if you specify the value of a previously defined sprite, the old sprite is replaced by the new sprite. If the old sprite had a *vertical-* or *horizontal-velocity* and you do not specify a new velocity, the new sprite retains the old velocity.

Character-code is a numeric expression with a value from 0-255, specifying the character that defines the sprite pattern.

If you use the MAGNIFY subprogram to change to double-sized sprites, the sprite definition includes the character specified by the character-code and three additional characters (see MAGNIFY).

Once defined by the SPRITE subprogram, the *character-code* of a sprite can be changed by the PATTERN subprogram.

The *foreground-color* is a numeric expression with a value from 1 to 16, specifying one of the 16 available colors. Once defined by the SPRITE subprogram, the *foreground-color* of a sprite can be changed by the COLOR subprogram.

The background-color of a sprite is always transparent.

The *pixel-row* and *pixel-column* are numeric expressions whose values specify the screen pixel position of the pixel at the upper-left corner of the sprite.

Once defined by the SPRITE subprogram, the *pixel-row* and *pixel-column* of a sprite can be changed by the LOCATE subprogram, and the current *pixel-row* and *pixel column* of a sprite can be ascertained by the POSITION subprogram. Also, the distance between sprites or between a sprite and a specified screen pixel can be ascertained by the DISTANCE subprogram, and the COINC subprogram can be used to ascertain whether sprites are coincident with each other or with a specified screen pixel.

4.105.5. Sprite Motion

Sprite motion is valid only in Pattern Mode. There is no effect in High-Resolution Mode.

The optional *vertical-velocity* and *horizontal-velocity* are numeric expressions with values from -128 to 127. If both values are zero, the sprite is stationary. The speed of a sprite is in direct linear proportion to the absolute value of the specified velocity.

A positive *vertical-velocity* causes the sprite to move toward the top of the screen; a negative vertical-velocity causes the sprite to move toward the bottom of the screen.

A positive *horizontal-velocity* causes the sprite to move to the right; a negative *horizontal-velocity* causes the sprite to move to the left.

If neither the *vertical-* nor *horizontal-velocity* are zero, the sprite moves at an angle, in a direction and at a speed determined by the velocity values.

The *velocity* of a sprite can be changed by the MOTION subprogram.

When a moving sprite reaches an edge of the screen, it disappears. The sprite reappears in the corresponding position at the opposite edge of the screen. The motion of a sprite may be affected by the computer's internal processing and by input to, and output from, external devices.

4.105.6. Program

The following three programs show some possible uses of sprites.

```
100 CALL CLEAR
110 CALL CHAR(244,"FFFFFFFFFFFFFFFF")
120 CALL CHAR(246,"183C7EFFFF7E3C18")
130 CALL CHAR(248,"F00FF00FF00FF00F")
140 CALL SPRITE(#1,244,5,92,124,#2,248,7,1,1)
150 CALL SPRITE(#28,33,16,12,48,1,1)
160 CALL SPRITE(#15,246,14,1,1,127,-128)
170 GOTO 170
```

(Press **CLEAR** to stop program.)

Line 140 creates a dark blue sprite in the center of the screen and a red striped sprite in the upper-right corner of the screen. Line 150 creates a white sprite near the upper-left corner of the screen and starts it moving slowly at a 45-degree angle down and to the right. The sprite is an exclamation point.

Line 160 creates a dark red sprite at the upper-right corner of the screen and starts it moving very fast at a 45 degree angle down and to the left.

The following program makes a rather spectacular use of sprites.

```
100 CALL CLEAR
110 CALL CHAR(244,"000808lC7FlC0808")
120 RANDOMIZE
130 CALL SCREEN(2)
140 FOR A=1 TO 28
150 CALL SPRITE(#A,244,INT(A/3)+3,92,124,A*INT(RND*4.5)
-2.25+A/2*SGN(RND-.5),A*INT(RND*4.5)-2.25+A/2*SGN(RND-.5))
160 NEXT A
170 GOTO 140
```

(Press **CLEAR** to stop the program.)

Line 110 defines character 244.

Line 150 defines the sprites, 28 in all. The sprite-number is the current value of A. The character-value is 244. The sprite-color is INT(A/3)+3. The starting dot-row and dot-column are 92 and 124, the center of the screen. The row- and column-velocities are chosen randomly using the value of A*INT(RND*4.5)-2.25+A/2*SGN(RND-.5).

Line 170 causes the sequence to repeat.

The following program uses all the subprogram that relate to sprites except for COLOR. They are CHAR, COINC, DELSPRITE, LOCATE, MAGNIFY, MOTION, PATTERN, POSITION, and SPRITE.

The program creates two double-sized magnified sprites in the shapes of two people walking along a floor. There is a barrier that one of them passes through and the other jumps through. The one that jumps through goes a little faster after each jump, eventually catching the other one. When this happens, they each become double-sized, unmagnified sprites and continue walking. When they meet for the second time, the one that has been going faster disappears and the other continues walking.

```
100 CALL CLEAR
110 S1$="0103030103030303030303030303030380C
0C080C0C0C0C0C0C0C0C0C0C0E011"
120 S2$="0103030103070FIBIB030303060C0C0E80C
0C080C0E0F0D8CCC0C0C060303038"
130 COUNT=0
140 CALL CHAR(244,S1$)
150 CALL CHAR(248,S2$)
160 CALL SCREEN(14)
170 CALL COLOR(14,13,13)
180 FOR A=19 TO 24
190 CALL HCHAR(A,1,136,32)
200 NEXT A
210 CALL COLOR(13,15,15)
220 CALL VCHAR(14,22,128,6)
230 CALL VCHAR(14,23,128,6)
```

```
240 CALL VCHAR(14,24,128,6)
250 CALL SPRITE(#1,244,5,113,129,#2,244,7,113,9)
260 CALL MAGNIFY(4)
270 XDIR=4
280 PAT=2
290 CALL MOTION(#1,0,XDIR,#2,0,4)
300 CALL PATTERN(#1,246+PAT,#2,246-PAT)
310 PAT=-PAT
320 CALL COINC(ALL,CO)
330 IF CO>0 THEN 370
340 CALL POSITION(#1,YPOS1,XPOS1)
350 IF XPOS1>136 AND XPOS1<192 THEN 470
360 GOTO 300
370 REM COINCIDENCE
380 CALL MOTION(#1,0,0#2,0,0)
390 CALL PATTERN(#1,244,#2,244)
400 IF COUNT>0 THEN 540
410 COUNT=COUNT+1
420 CALL POSITION(#1,YPOS1,XPOS1,#2,YPOS2,XPOS2)
430 CALL MAGNIFY(3)
440 CALL LOCATE(#1,YPOS1+16,XPOS1+8,#2,YPOS2+16,XPOS2)
450 CALL MOTION(#1,0,XDIR,#2,0,4)
460 GOTO 340
470 REM #1 HIT WALL
480 CALL MOTION(#1,0,0)
490 CALL POSITION(#1,YPOS1,XPOS1)
500 CALL LOCATE(#1,YPOS1,193)
510 XDIR=XDIR+1
520 CALL MOTION(#1,0,XDIR)
530 GOTO 300
540 REM SECOND COINCIDENCE
550 FOR DELAY=1 TO 1000 :: NEXT DELAY
560 CALL MOTION(#2,0,4)
570 CALL DELSPRITE(#1)
580 FOR STEP1=1 TO 20
590 CALL PATTERN(#2,248)
600 FOR DELAY=1 TO 40 :: NEXT DELAY
610 CALL PATTERN(#2,244)
620 FOR DELAY=1 TO 40 :: NEXT DELAY
630 NEXT STEP1
640 CALL CLEAR
```

Lines 110, 120, 140, 150, 250, and 260 define the sprites.

Line 130 sets the meeting counter to zero.

Lines 170 through 200 build the floor.

Lines 210 through 240 build the barrier.

Line 270 sets the starting speed of the sprite that will speed up.

Line 290 sets the sprites in motion.

Line 300 creates the illusion of walking.

Line 320 checks to see if the sprites have met. Line 330 transfers control if the sprites have met. Lines 340 and 350 check to see if the sprite has reached the barrier and transfer control if it has.

Line 360 loops back to continue the walk. Lines 370 through 460 handle the sprites running into each other. Lines 380 and 390 stop them.

Line 400 checks to see if it is the first meeting. Line 410 increments the meeting counter. Line 420 finds the sprites position.

Line 430 makes them smaller. Line 440 puts them on the floor and moves the fast one slightly ahead. Line 450 starts them moving again.

Lines 470 through 530 handle the fast sprite jumping through the barrier. Line 480 stops it. Line 490 finds where it is.

Line 500 puts it at the new location beyond the barrier. Lines 510 and 520 start it moving again, a little faster.

Lines 540 through 640 handle the second meeting.

Line 560 starts the slow sprite moving. Line 570 deletes the fast sprite.

Lines 580 through 630 make the slow sprite walk 20 steps.

4.106. SQR function — Square Root

4.106.1. Format

`SQR(`*`numeric-expression`*`)`

4.106.2. Type

REAL

4.106.3. Description

The SQR function returns the positive square root of the value of the *numeric-expression*.

The value of the *numeric-expression* cannot be negative.

4.106.4. Examples

```
100 PRINT SQR(4)
```

Prints 2.

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

Sets X equal to the square root of 257,000, which is 506.0516742255.

4.107. STOP

4.107.1. Format

```
STOP
```

4.107.2. Cross Reference

END

4.107.3. Description

The STOP statement stops the execution of your program.

When your computer encounters a STOP statement, the computer performs the following operations:

- It closes all open files.
- If the computer is in Pattern or Text Mode, it restores the default character definitions of all characters.
- If the computer is in High-Resolution Mode, it restores the default character definitions of all characters and restores the default graphics mode (Pattern) and margin settings (3, 30, 1, 24)
- Restores the default foreground color (black) and background color (transparent) to all characters.
- Restores the default screen color (cyan).
- Deletes all sprites.
- 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.

A STOP statement is not necessary to stop your program; the program automatically stops after the highest-numbered line is executed.

STOP is frequently used before a subprogram that follows the main portion of a program, to ensure that the subprogram is not executed after the execution of the highest-numbered line in the main program.

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

4.107.4. Program

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

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

4.108. STR$ function— String-Number

4.108.1. Format

`STR$(`*`numeric-expression`*`)`

4.108.2. Type

String

4.108.3. Cross Reference

VAL

4.108.4. Description

The STR$ function returns the string representation of the value of the *numeric-expression*.

STR$ enables you to use the string representation of the *numeric-expression* with an instruction that requires a string expression as a parameter.

STR$ is the inverse of the VAL function.

STR$ removes leading and trailing spaces.

4.108.5. Examples

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

Sets NUM$ equal to "78.6".

```
100 LL$=STR$(3El5)
```

Sets LL$ equal to 113.E+15".

```
100 X$=STR$(A*4)
```

Sets X$ equal to a string representation of whatever value is obtained when A is multiplied by 4. For instance, if A is equal to -8, X$ is set equal to "-32".

4.109. SUB — Subprogram

4.109.1. Format

`SUB` *`subprogram-name`*`[([`*`data-type`*` ]`*`parameter`*`[,...])]`

4.109.2. Cross Reference

CALL, SUBEND, SUBEXIT

4.109.3. Description

The SUB statement is the first statement in a subprogram.

You can use a subprogram to separate a group of statements from the main program. Subprogram are generally used to perform a specific operation several times in the same program or in different programs, or to isolate variables that are specific to the subprogram.

Subprogram are accessed from your main program with a CALL statement. The *subprogram-name* in the SUB statement is the same name that you use in the CALL statement that transfers control to the subprogram.

The maximum length of a *subprogram-name* is 15 characters.

A user-written subprogram may have the same subprogram-name as a built-in subprogram. In such a case, a CALL statement will access the user-written subprogram instead of the built-in one.

You can use *parameters* to pass values to a subprogram. *Parameters* must be valid names of variables or arrays.

If a parameter is numeric, you can optionally specify its *data-type* (DEFINT or REAL). Numeric parameters are considered to be REAL unless you specifically declare them as DEFINT in the parameter list. A DEFINT ALL, statement in your main program or in the subprogram itself does not affect the *data-type* of *parameters*.

SUBEND must he the last statement executed in a subprogram. When the computer encounters a SUBEND or a SUBEXIT statement in a subprogram, program control returns to the statement immediately following the CALL statement that called the subprogram.

It is recommended that you do not use any statement other than SUBEND or SUBEXIT to leave a subprogram. If you use another statement to leave a subprogram you may still be using variables local to the subprogram, which may cause unexpected results.

Subprogram must have higher line numbers than any part of your main program. A SUB statement cannot be part of an IF THEN statement.

4.109.4. Subprogram Variables

The variables used in a subprogram (other than those used as parameters) are local to the subprogram; that is, even if a variable in your main program has the same name as a variable in a subprogram, the value of that variable outside the subprogram is not affected by changes to its value in the subprogram. If a subprogram is called more than once, any local variables used in the subprogram retain their values from one call to the next.

Numeric variables used in a subprogram are considered to be REAL unless you specifically declare them as DEFINT. A DEFINT ALL statement in your main program does not affect the data-type of numeric variables in a subprogram.

A DEFINT ALL statement in a subprogram does not affect the data-type of numeric variables in your main program or in any other subprogram.

4.109.5. Parameters

When your program executes a subprogram beginning with a SUB statement with parameters, the parameter values (constants or variables) are passed from the *parameter-list* of the CALL statement to the subprogram. The*parameter-list* in the CALL statement must contain the same number of parameters as the SUB statement. Values are passed in the order in which they are listed.

A numeric parameter must be passed a numeric value. A string parameter must be passed a string value.

An array parameter must be passed an array. A string-array parameter must be passed a string array; a DEFINT-array parameter must be passed a DEFINT array; a REAL-array parameter must be passed a REAL array.

To pass an entire array as one parameter, follow the array name with left and right parentheses. If the array has more than one dimension, place one comma between the parentheses for each additional dimension.

4.109.6. Passing Parameters by Reference and Value

When a subprogram manipulates the value of a parameter passed to it, the new parameter value may or may not be passed back to the main program. When a parameter is passed to a subprogram "by reference", the new value is passed back to the main program after the subprogram has executed.

When a parameter is passed to a subprogram "by value", the new value is not passed back to the main program.

Variables, array elements, and arrays are normally passed by reference. However, if a numeric variable or array element is of a different data-type in the main program than it is in the subprogram (DEFINT vs. REAL), the parameter is passed by value. Note that if you pass a DEFINT variable to a numeric parameter, that subprogram parameter is considered to be REAL unless you specifically declare it as a DEFINT by using the data-type option in the parameter list of the SUB statement. Remember that if you want to pass a DEFINT by reference, you must declare the subprogram parameter as a DEFINT.

To specify that a variable or array element is to be passed by value rather than by reference, enclose it in parentheses in the CALL statement's parameter-list. Note that this option is not available for arrays.

If you use an expression as a parameter, it is evaluated and passed by value.

4.109.7. Examples

```
100 SUB MENU
```

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

```
100 SUB MENU(COUNT,CHOICE)
```

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

```
100 SUB PAYCHECK(DATE,Q,SSN,PAYRATE,TABLE(,))
```

Marks the beginning of a subprogram. The variables DATE, Q, SSN, PAYRATE, and the array TABLE with two dimensions may be used and/or have their values changed in the subprogram and returned to the variables in the same position in the calling statement.

4.109.8. Program

The following program illustrates the use of SUB. The subprogram MENU had been previously saved with the MERGE option. It prints a menu and requests a choice. The main program tells the subprogram how many choices there are and what the choices are. It then uses the choice made in the subprogram to determine what program to run.

```
100 CALL MENU(5,R)
110 ON R GOTO 12(,130,140,150,160)
120 RUN "DSK1.PAYABLES"
130 RUN "DSK1.RECEIVE"
140 RUN "DSK1.PAYROLL"
150 RUN "DSK1.INVENTORY"
160 RUN "DSK1.LEDGER"
170 DATA ACCOUNTS PAYABLE,ACCOUNTS RECEIVABLE,PAYROLL,INVENTORY,GENERAL LEDGER
```

Beginning of subprogram MENU.

Note that this R is not the same as the R used in lines 100 and 110 in the main program.

```
10000 SUB MENU(COUNT,CHOICE)
10010 CALL CLEAR
10020 IF COUNT>22 THEN PRINT "TOO MANY ITEMS" :: CHOICE=0 :: SUBEXIT
10030 RESTORE
10040 FOR R=1 TO COUNT
10050 READ TEMP$
10060 TEMP$=SEG$(TEMP$,1,25)
10070 DISPLAY AT(R,1):R;TEMP$
10080 NEXT R
10090 DISPLAY AT(R+1,1):"YOUR CHOICE: I"
10100 ACCEPT AT(R+1,14)BEEP VALIDATE(DIGIT)SIZE(-2):CHOICE
10110 IF CHOICE>COUNT OR CHOICE<1 THEN 10100
10120 SUBEND
```

4.110. SUBEND — Subprogram End

4.110.1. Format

```
SUBEND
```

4.110.2. Cross Reference

SUB, SUBEXIT

4.110.3. Description

The SUBEND statement marks the end of a subprogram.

SUBEND must be the last statement executed in a subprogram. When the computer encounters a SUBEND statement in a subprogram, program control returns to the statement immediately following the CALL statement that called the subprogram.

It is recommended that you do not use any statement other than SUBEND or SUBEXIT to leave a subprogram. If you use another statement to leave a subprogram you may still be using variables local to the subprogram, which may cause unexpected results.

A SUBEND statement cannot be part of an IF THEN statement.

The only statements that can immediately follow a SUBEND statement are REM, END, or the SUB statement for the next subprogram.

4.111. SUBEXIT — Subprogram Exit

4.111.1. Format

```
SUBEXIT
```

4.111.2. Cross Reference

SUB, SUBEND

4.111.3. Description

The SUBEXIT statement enables you to leave a subprogram before the computer executes the SUBEND statement that ends the subprogram.

SUBEXIT enables you to have more than one exit from a subprogram.

When the computer encounters a SUBEXIT statement in a subprogram, program control returns to the statement immediately following the CALL statement that called the subprogram.

It is recommended that you do not use any statement other than SUBEND or SUBEXIT to leave a subprogram. If you use another statement to leave a subprogram you may still be using variables local to the subprogram, which may cause unexpected results.

4.112. TAB function — Tabulate

4.112.1. Format

`TAB(`*`numeric-expression`*`)`

4.112.2. Cross Reference

DISPLAY, PRINT

4.112.3. Description

The TAB function specifies the starting position of the next item to be printed by a PRINT or DISPLAY instruction.

The *numeric-expression* specifies the starting position of the next print item in a print-list of a PRINT or DISPLAY instruction.

If the value of the *numeric-expression* is not an integer, it is rounded to the nearest integer. If the value of the *numeric-expression* is less than 1, it is replaced by 1.

If the value of the *numeric-expression* is greater than the record length of the screen or device, it is repeatedly reduced by the record length until it is less than or equal to the record length. The record length of the screen is the width of the screen window defined by the margins. For more information about the record length of a particular device, refer to the owner's manual that comes with that device.

TAB is relative to the left side of the screen, not the current screen window.

Because the TAB function itself is treated as a separate print item, it must be preceded and/or followed by a print separator (usually a semicolon), unless it is the only item in the print-list.

If the number of characters already printed in the current record is greater than or equal to the position indicated by the value of the *numeric-expression*, the print item following the TAB is printed in the next record, beginning in the position specified by the value of the *numeric-expression*.

TAB can be used to print to a device or file only if the device or file has been opened in DISPLAY format.

TAB cannot be used with PRINT USING or DISPLAY USING.

4.112.4. Examples

```
100 PRINT TAB(12);35
```

Prints the number 35 at the twelfth position.

```
100 PRINT 356;TAB(18);"NAME"
```

Prints 356 at the beginning of the line and NAME at the eighteenth position of the line.

```
100 PRINT "ABCDEFGHIJKLM";TAB(5);"NOP"
```

Prints ABCDEFGHIJKLM at the beginning of the line and NOP at the fifth position of the next line.

```
100 DISPLAY AT(12,1):"NAME";TAB(15);"ADDRESS"
```

Displays NAME at the beginning of the twelfth line on the screen and ADDRESS at the fifteenth position on the twelfth line of the screen.

4.113. TAN function — Tangent

4.113.1. Format

`TAN(`*`numeric-expression`*`)`

4.113.2. Type

REAL

4.113.3. Cross Reference

ATN, COS, SIN

4.113.4. Description

The TAN function returns the tangent of the angle whose measurement in radians is the value of the *numeric-expression*.

The *numeric-expression* cannot be less than -1.5707963269514E10 or greater than 1.5707963266374E10.

To convert the measure to radians, multiply by pi/180.

4.113.5. Program

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

```
100 A=.7853981633973
110 B=26.565051177
120 C=45*PI/180
130 PRINT TAN(A);TAN(B)
140 PRINT TAN(B*PI/180)
150 PRINT TAN(C)
RUN
1. 7.17470553
.5
1
```

4.114. TERMCHAR function — Termination Character

4.114.1. Format

```
TERMCHAR
```

4.114.2. Type

DEFINT

4.114.3. Cross Reference

ACCEPT, INPUT, LINPUT

4.114.4. Description

The TERMCHAR function returns the character code of the key pressed to exit from the previously executed INPUT, ACCEPT, or LINPUT statement.

In a program, the value returned by TERMCHAR depends on the key pressed to exit from the last instruction that accepted input from the keyboard.

VALUE RETURNED	*KEY*
1	**AID**
2	**CLEAR**
10	**DOWN ARROW**
11	**UP ARROW**
12	**PROC'D**
13	**ENTER**
14	**BEGIN**
15	**BACK**

If you use TERMCHAR as part of a command (unless it is preceded by ACCEPT, INPUT, or LINPUT), the value returned depends on the key pressed to enter the command (**ENTER**, **UP ARROW**, or **DOWN ARROW**).

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

4.114.5. Program

The following program illustrates a use of TERMCHAR. The program displays name, address, and city, state, and zip code information entered from the keyboard. Line 160 enables you to correct errors in previously entered lines by pressing **UP ARROW**. This returns the cursor to the beginning of the line that immediately precedes the one from which **UP ARROW** was entered.

```
100 CALL CLEAR
110 R=5*:C=12
120 DISPLAY AT(R,C-10):"NAME
130 DISPLAY AT(R+1,C-10):"ADDRESS:"
140 DISPLAY AT(R+2,C-10):"C,S,Z:"
150 ACCEPT AT(R,C)SIZE(-20):A$(R)
160 IF TERMCHAR=11 THEN R=R-1 ELSE R=R+1
170 IF R=7 THEN 150
180 DISPLAY AT(20,1):A$(5):A$(6):A$(7)
190 GOTO 110
```

(Press **CLEAR** to stop the program.)

4.115. TRACE

4.115.1. Format

```
TRACE
```

4.115.2. Cross Reference

UNTRACE

4.115.3. Description

The TRACE instruction causes the computer to display the line number of each line in your program before it is executed.

TRACE enables you to see the order in which the computer performs statements as it runs your program. It is valuable as a debugging aid to help you find errors (such as unwanted infinite loops) in your program.

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

The effect of a TRACE instruction is canceled when an UNTRACE instruction or a NEW command is executed.

4.115.4. Programs

The following program displays a trace of the order of execution of the program lines.

```
100 FOR J=1 TO 3
110 PRINT "WORD"
120 NEXT J
TRACE

<100><110> WORD
<120><110> WORD
<120><110> WORD
<120>
```

4.116. UNBREAK

4.116.1. Format

```
UNBREAK [line-number-list]
```

4.116.2. Cross Reference

BREAK

4.116.3. Description

The UNBREAK instruction removes a breakpoint from each program statement you specify.

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

The *line-number-list* consists of one or more line numbers separated by commas. When an UNBREAK instruction is executed, breakpoints are removed from the specified program lines.

If you do not include a *line-number-list*, UNBREAK removes all breakpoints, except for a breakpoint that occurs when a BREAK statement with no *line-number-list* is encountered in a program.

If the *line-number-list* includes an invalid line number (0 or a value greater than 32767), the message `BAD LINE NUMBER` is displayed. If the *line-number-list* includes a fractional or negative line number, the message `SYNTAX ERROR` is displayed. In both cases, the UNBREAK instruction is ignored; that is, breakpoints are not removed even at valid line numbers in the *line-number-list*. If you were entering UNBREAK as a program statement, it is not entered into your program.

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

```
* WARNING
  LINE NOT FOUND
```

is displayed. (If you were entering UNBREAK as a program statement, the line-number is included in the warning message.) A breakpoint is, however, removed from any valid line in the *line-number-list* that precedes the line number that caused the warning.

4.116.4. Examples

```
UNBREAK
450 UNBREAK
```

Removes all breakpoints (except those resulting from a BREAK statement with no line-number-list).

```
UNBREAK 100,130
350 UNBREAK 100,130
```

Removes the breakpoints from lines 100 and 130.

4.117. UNTRACE

4.117.1. Format

```
UNTRACE
```

4.117.2. Cross Reference

TRACE

4.117.3. Description

The UNTRACE instruction cancels the effect of a TRACE instruction.

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

4.117.4. Examples

```
UNTRACE
450 UNTRACE
```

Removes the effect of TRACE.

4.118. VAL function — Value

4.118.1. Format

`VAL(`*`string-expression`*`)`

4.118.2. Type

REAL

4.118.3. Cross Reference

STR$

4.118.4. Description

The VAL function returns the numeric value of the *string-expression*.

VAL enables you to use the numeric value of the *string-expression* with an instruction that requires a numeric expression as a parameter.

VAL is the inverse of the STR$ function.

The *string-expression* must be a valid representation of a number. The length of the *string-expression* must be greater than 0 and less than 255. If you use a string constant, it must be enclosed in quotation marks.

4.118.5. Example

```
100 NUMB=VAL("78.6")
110 PRINT NUMB
```

Prints 78.6.

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

Sets LL equal to 3E+15, or 315.

4.119. VALHEX function — Value of Hexadecimal Number

4.119.1. Format

`VALHEX(`*`string-expression`*`)`

4.119.2. Type

DEFINT

4.119.3. Description

VALHEX returns the numeric value of the hexadecimal number represented by the *string-expression*.

The *string-expression* specifies the hexadecimal (base 16) number to be converted to a decimal (base 10) number. If you use a string constant, it must be enclosed in quotation marks.

The *string-expression* must contain only valid hexadecimal digits (0-9,A-F). Alphabetic hexadecimal digits must be upper-case letters. VALHEX can convert a hexadecimal number from one to four digits long. If the length of the *string-expression* is greater than four, VALHEX uses only the last four characters.

VALHEX returns an integer greater than or equal to -32768 (hexadecimal 8000) and less than or equal to 32767 (hexadecimal 7FFF).

4.119.4. Examples

```
100 A=VALHEX("400A")
```

Sets A equal to 16394.

```
100 PRINT VALHEX("8200")
```

Prints -32256.

4.120. VCHAR subprogram — Vertical Character

4.120.1. Format

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

4.120.2. Cross Reference

DCOLOR, GCHAR, GRAPHICS, HCHAR

4.120.3. Description

The VCHAR 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, *row* must be less than or equal to 24.

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

VCHAR 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 column, they continue from the first character of the next column. If the repetitions extend past the end of the last column, they continue from the first character of the first column.

If you use VCHAR to display a character on the screen, and then later use CHAR, 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 High-Resolution Mode, the displayed character remains unchanged.

4.120.4. Examples

```
100 CALL VCHAR(12,16,33)
```

Places character 33 (an exclamation point) in row 12, column 16.

```
100 CALL VCHAR(1,1,ASC("!"),768)
```

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

```
100 CALL VCHAR(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.

4.121. VERSION subprogram

4.121.1. Format

```
CALL VERSION(numeric-variable)
```

4.121.2. Description

The VERSION subprogram returns a value indicating the version of BASIC being used.

In MYARC Extended BASIC II, VERSION returns a value of 200 to the *numeric-variable* you specify.

4.121.3. Example

```
100 CALL VERSION(V)
```

Sets V equal to 200.

4.122. WRITE subprogram

4.122.1. Format

```
CALL WRITE(type,row,column,string-expression
[,row2,column2,string-expression2[,...]])
```

4.122.2. Cross Reference

GRAPHICS, HCHAR, MARGIN, VCHAR

4.122.3. Description

The WRITE subprogram enables you to display strings on the screen in High-Resolution Mode. The *string-expression* may be a constant or a variable.

Type is a numeric-expression whose value specifies the action taken by the WRITE subprogram.

TYPE	*ACTION*
0, 1	Displays the *string-expression* horizontally within the screen boundaries. Margins are disregarded. Display begins at the specified *row* and *column*.
2	Displays the *string-expression* vertically within the screen boundaries. Margins are disregarded. Display begins at the specified *row* and *column*.

If the string to be displayed will not fit within the screen, the string will wrap around on the screen.

For WRITE, the screen is considered to be 24 rows by 32 columns. As in HCHAR and VCHAR, blocks of pixels 8x8 are the unit of measurement not single pixels as in most other High-Resolution subprogram.

Row must have a value from 1 to 24, *column* must have a value from 1 to 32.

WRITE can only be used in High-Resolution Mode. An error results if you use WRITE in Pattern or Text Modes.

4.122.4. Example

```
100 CALL GRAPHICS(3)
110 CALL WRITE(0,1,1,"HELLO, HOW ARE YOU")
```

Displays "HELLO, HOW ARE YOU", starting at row 1, column 1, relative to the upper-left corner of the screen.

5. APPENDICES

The following appendices give useful information for utilization with MYARC Extended BASIC II.

Appendix A: List of Commands, Statements, and Functions

Appendix B: ASCII Codes

Appendix C: Musical Tone Frequencies

Appendix D: Character Sets

Appendix E: Pattern-Identifier Conversion Table

Appendix F: Color Codes

Appendix G: Mathematical Functions

Appendix H: List of Speech Words

Appendix I: Adding Suffixes to Speech Words

Appendix J: Error Messages

Appendix K: High-Resolution Mode (Restrictions and Conventions)

Appendix A. Commands, Statements, and Functions

The following is a list of all MYARC Extended BASIC II commands, statements, and functions. Commands are listed first; if a command can also be used as a statement, the letter "S" is listed to the right of the command. Commands that can be abbreviated have the acceptable abbreviations underlined. Next is a list of all MYARC Extended BASIC II statements; those that can also be used as commands have a "C" after them.

Finally, there is a list of all MYARC Extended BASIC II functions.

MYARC Extended BASIC II COMMANDS

BREAK S
BYE
CONTINUE
DELETE S
INTEGER S
LIST
MERGE
NEW
NUMBER
OLD
RESEQUENCE
RUN S
SAVE
SIZE
TRACE S
UNBREAK S
UNTRACE S

MYARC Extended BASIC II STATEMENTS

ACCEPT	C
CALL	C
CALL CHAR	C
CALL CHARPAT	C
CALL CHARSET	C
CALL CLEAR	C
CLOSE	C
CALL COINC	C
CALL COLOR	C
DATA	
DEF	
CALL DELSPRITE	C
DIM	C
DISPLAY	C
DISPLAY USING	C
CALL DISTANCE	C
END	
CALL ERR	C
FOR	C
CALL GCHAR	C
GOSUB	
GOTO	
CALL GRAPHICS	C
CALL HCHAR	C
IF THEN ELSE	
IMAGE	
CALL INIT	C
INPUT	
INPUT REC	
CALL JOYST	C
CALL KEY	C
[LET]	C
CALL LINK	C
LINPUT	
CALL LOAD	C
CALL LOCATE	C
CALL MAGNIFY	C
CALL MOTION	C
NEXT	C
ON BREAK	
ON ERROR	
ON GOSUB	
ON GOTO	
ON WARNING	
OPEN	C
CALL PEEKV	S
CALL POKEV	S
REAL	S
OPTION BASE	
CALL PATTERN	C
CALL PEEK	C
CALL POSITION	C
PRINT	C
PRINT USING	C
RANDOMIZE	C
READ	C
REM	C
RESTORE	C
RETURN	
CALL SAY	C
CALL SCREEN	C
CALL SOUND	C
CALL SPGET	C
CALL SPRITE	C
STOP	C
SUB	
SUBEND	
SUBEXIT	
CALL VCHAR	C
CALL VERSION	C

MYARC Extended BASIC II FUNCTIONS

ABS
ASC
ATN
CHR$
COS
EOF
EXP
FREESPACE
INT

LEN
LOG
MAX
MIN
PI
POS
REC
RND
RPT$

SEG$
SGN
SIN
SQR
STR$
TAB
TAN
TERMCHAR
VAL
VALHEX

Appendix B. ASCII Codes

The following predefined characters may be printed or displayed on the screen.

Ascii		*Ascii*	
Code	*Character*	*Code*	*Character*
30	(cursor)	79	O
31	(edge character)	80	P
32	(space)	81	Q
33	! (exclamation point)	82	R
34	" (quote)	83	S
35	# (number or pound sign)	84	T
36	$ (dollar)	85	U
37	% (percent)	86	V
38	& (ampersand)	87	W
39	' (apostrophe)	88	X
40	((open parenthesis)	89	Y
41	) (close parenthesis)	90	Z
42	* (asterisk)	91	[(open bracket)
43	+ (plus)	92	\ (reverse slash)
44	, (comma)	93	] (close bracket)
45	- (minus)	94	^ (exponentiation)
46	. (period)	95	_ (underline)
47	/ (slash)	96	` (grave)
48	0	97	a
49	1	98	b
50	2	99	c
51	3	100	d
52	4	101	e
53	5	102	f
54	6	103	g
55	7	104	h
56	8	105	i
57	9	106	j
58	: (colon)	107	k
59	; (semicolon)	108	l
60	< (less than)	109	m
61	= (equals)	110	n
62	> (greater than)	111	o
63	? (question mark)	112	p
64	@ (at sign)	113	q
65	A	114	r
66	B	115	s
67	C	116	t
68	D	117	u

69	E	118	v
70	F	119	w
71	G	120	x
72	H	121	y
73	I	122	z
74	J	123	{ (left brace)
75	K	124	| (vertical bar)
76	L	125	} (right brace)
77	M	126	~ (tilde)
78	N	127	DEL (appears as a blank)

The following key presses may also be detected by CALL KEY.

1	**FCTN 7** (AID)	10	**FCTN X** (DOWN ARROW)
3	**FCTN 1** (DEL)	11	**FCTN E** (UP ARROW)
4	**FCTN 2** (INS)	12	**FCTN 6** (PROC'D)
6	**FCTN 8** (REDO)	13	**ENTER**
7	**FCTN 3** (ERASE)	14	**FCTN 5** (BEGIN)
8	**FCTN S** (LEFT ARROW)	15	**FCTN 9** (BACK)
9	**FCTN D** (RIGHT ARROW)		

Appendix C. Musical Tone Frequencies

The following table gives the frequencies (rounded to integers) of four octaves of the tempered scale (one half step between notes). While this list does not represent the entire range of tones that the computer can produce, it can be helpful for programming music.

FREQUENCY	*NOTE*	*FREQUENCY*	*NOTE*
110	A	440	A (above middle C)
117	A#, Bb	466	A#, Bb
123	B	494	B
131	C (low C)	523	C (high C)
139	C# Db	554	C#, Db
147	D	587	D
156	D#, Eb	622	D#, Eb
165	E	659	E
175	F	698	F
185	F#, Gb	740	F#, Gb
196	G	784	G
208	G#, Ab	831	G#, Ab
220	A (below middle C)	880	A (above high C)
220	A (below middle C)	880	A (above high C)
233	A#, Bb	932	A#, Bb
247	B	988	B
262	C (middle C)	1047	C
277	C#, Db	1109	C#, Db
294	D	1175	D
311	D#, Eb	1245	D#, Eb
330	E	1319	E
349	F	1397	F
370	F#, Gb	1480	F#, Gb
392	G	1568	G
415	G#, Ab	1661	G#, Ab
440	A (above middle C)	1760	A

Appendix D. Character Sets

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

Appendix E. Pattern Identifier Conversion Table

Blocks				Binary Code (0 = off; 1 = on)	Hexadecimal Code
				0000	0
			■	0001	1
		■		0010	2
		■	■	0011	3
	■			0100	4
	■		■	0101	5
	■	■		0110	6
	■	■	■	0111	7
■				1000	8
■			■	1001	9
■		■		1010	A
■		■	■	1011	B
■	■			1100	C
■	■		■	1101	D
■	■	■		1110	E
■	■	■	■	1111	F

Appendix F. Color Codes

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

Appendix G. Mathematical Functions

The following mathematical functions may be defined with DEF as shown.

Function	*MYARC Extended BASIC II statement*
Secant	DEF SEC(X)=1/COS(X)
Cosecant	DEF CSC(X)=1/SIN(X)
Cotangent	DEF COT(X)=1/TAN(X)
Inverse Sine	DEF ARCSIN(X)=ATN(X/SQR(1-X*X))
Inverse Cosine	DEF ARCCOS(X)=-ATN(X/SQR(1-X*X))+PI/2
Inverse Secant	DEF ARCSEC(X)=ATN(SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse Cosecant	DEF ARCCSC(X)=ATN(1/SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse Cotangent	DEF ARCCOT(X)=PI/2-ATN(X) or =PI/2+ATN(-X)
Hyperbolic Sine	DEF SINH(X)=(EXP(X)-EXP(-X))/2
Hyperbolic Cosine	DEF COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic Tangent	DEF TANH(X)=-2*EXP(-X)/(EXP(X)+EXP(-X))+1
Hyperbolic Secant	DEF SECH=2/(EXP(X)+EXP(-X))
Hyperbolic Cosecant	DEF CSCH=2/(EXP(X)-EXP(-X))
Hyperbolic Cotangent	DEF COTH(X)=2*EXP(-X)/(EXP(X)-EXP(-X))+1
Inverse Hyperbolic Sine	DEF ARCSINH(X)=LOG(X+SQR(X*X+1))
Inverse Hyperbolic Cosine	DEF ARCCOSH(X)=LOG(X+SQR(X*X-1))
Inverse Hyperbolic Tangent	DEF ARCTANH(X)=LOG((1+X)/(1-X))/2
Inverse Hyperbolic Secant	DEF ARCSECH(X)=LOG((1+SQR(1-X*X))/X)
Inverse Hyperbolic Cosecant	DEF ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
Inverse Hyperbolic Cotangent	DEF ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix H. List of Speech Words

The following is a list of all the letters, numbers, words, and phrases that can be accessed with CALL SAY and CALL SPGET. See *Appendix I* for instructions on adding suffixes to anything in this list.

- (NEGATIVE)
+ (POSITIVE)
. (POINT)
0
1
2
3
4
5
6
7
8
9
A (a)
A1 (^e)
ABOUT
AFTER
AGAIN
ALL
AM
AN
AND
ANSWER
ANY
ARE
AS
ASSUME
AT
B
BACK
BASE
BE
BETWEEN
BLACK
BLUE
BOTH
BOTTOM
BUT
BUY
BY
BYE
C
CAN
CASSETTE
CENTER
CHECK
CHOICE
CLEAR
COLOR
COME
COMES
COMMA
COMMAND
COMPLETE
COMPLETED
COMPUTER
CONNECTED
CONSOLE
CORRECT
COURSE
CYAN
D
DATA
DECIDE
DEVICE
DID
DIFFERENT
DISKETTE
DO
DOES
DOING
DONE
DOUBLE
DOWN
DRAW
DRAWING
E
EACH
EIGHT
EIGHTY
ELEVEN
ELSE
END
ENDS
ENTER
ERROR
EXACTLY
EYE
F
FIFTEEN
FIFTY
FIGURE
FIND
FINE
FINISH
FINISHED
FIRST
FIT
FIVE
FOR
FORTY
FOUR
FOURTEEN
FOURTH
FROM
FRONT
G
GAMES
GET
GETTING
GIVE
GIVES
GO
GOES
GOING
GOOD
GOOD WORK
GOODBYE
GOT
GRAY

GREEN
GUESS
H
HAD
HAND
HANDHELD UNIT
HAS
HAVE
HEAD
HEAR
HELLO
HELP
HERE
HIGHER
HIT
HOME
HOW
HUNDRED
HURRY
I
I WIN
IF
IN
INCH
INCHES
INSTRUCTION
INSTRUCTIONS
IS
IT
J
JOYSTICK
JUST
K
KEY
KEYBOARD
KNOW
L
LARGE
LARGER
LARGEST
LAST
LEARN
LEFT
LESS
LET
LIKE
LIKES
LINE
LOAD
LONG
LOOK
LOOKS
LOWER
M
MADE
MAGENTA
MAKE
ME
MEAN
MEMORY
MESSAGE
MESSAGES
MIDDLE
MIGHT
MODULE
MORE
MOST
MOVE
MUST
N
NAME
NEAR
NEED
NEGATIVE
NEXT
NICE TRY
NINE
NINETY
NO
NOT
NOW
NUMBER
O
OF
OFF
OH
ON
ONE
ONLY
OR
ORDER
OTHER
OUT
OVER
P
PART
PARTNER
PARTS
PERIOD
PLAY
PLAYS
PLEASE
POINT
POSITION
POSITIVE
PRESS
PRINT
PRINTER
PROBLEM
PROBLEMS
PROGRAM
PUT
PUTTING
Q
R
RANDOMLY
READ (read)
READ1 (red)
READY TO START
RECORDER
RED
REFER
REMEMBER
RETURN
REWIND
RIGHT
ROUND
S
SAID
SAVE
SAY
SAYS
SCREEN
SECOND
SEE

SEES
SET
SEVEN
SEVENTY
SHAPE
SHAPES
SHIFT
SHORT
SHORTER
SHOULD
SIDE
SIDES
SIX
SIXTY
SMALL
SMALLER
SMALLEST
SO
SOME
SORRY
SPACE
SPACES
SPELL
SQUARE
START
STEP
STOP
SUM
SUPPOSED
SUPPOSED TO
SURE
T
TAKE
TEEN
TELL
TEN
TEXAS INSTRUMENTS
THAN
THAT
THAT IS INCORRECT
THAT IS RIGHT
THE (the)
THE1 (th^e)
THEIR
THEN
THERE
THESE
THEY
THING
THINGS
THINK
THIRD
THIRTEEN
THIRTY
THIS
THREE
THREW
THROUGH
TIME
TO
TOGETHER
TONE
TOO
TOP
TRY
TRY AGAIN
TURN
TWELVE
TWENTY
TWO
TYPE
U
UHOH
UNDER
UNDERSTAND
UNTIL
UP
UPPER
USE
V
VARY
VERY
W
WAIT
WANT
WANTS
WAY
WE
WEIGH
WEIGHT
WELL
WERE
WHAT
WHAT WAS THAT
WHEN
WHERE
WHICH
WHITE
WHO
WHY
WILL
WITH
WON
WORD
WORDS
WORK
WORKING
WRITE
X
Y
YELLOW
YES
YET
YOU
YOU WIN
YOUR
Z
ZERO

Appendix I. Adding Suffixes to Speech Words

This appendix describes how to add ING, S, and ED to any word available in the *Solid State Speech*™ Synthesizer resident vocabulary.

The code for a word is first read using SPGET. The code consists of a number of characters, one of which tells the speech unit the length of the word. Then, by means of the subprogram listed here, additional codes can be added to give the sound of a suffix.

Words often have trailing-off data that make the word sound more natural but prevent the easy addition of suffixes. In order to add suffixes this trailing-off data must be removed.

The following program allows you to input a word and, by trying different truncation values, make the suffix sound like a natural part of the word. The subprogram DEFING (lines 1000 through 1130), DEFS1 (lines 2000 through 2100), DEFS2 (lines 3000 through 3090), DEFS3 (lines 4000 through 4120), DEFED1 (lines 5000 through 5070), DEFED2 (lines 6000 through 6110), DEFED3 (lines 7000 through 7130), and MENU (lines 10000 through 10120) should be input separately and saved with the MERGE option. (The subprogram MENU is the same one used in the illustrative program with SUB.) You may wish to use different line numbers. Each of these subprogram (except MENU) defines a suffix.

DEFING defines the ING sound. DEFS1 defines the S sound as it occurs at the end of "cats". DEFS2 defines the S sound as it occurs at the end of "cads". DEFS3 defines the S sound as it occurs at the end of "wishes". DEFED1 defines the ED sound as it occurs at the end of "passed". DEFED2 defines the ED sound as it occurs at the end of "caused". DEFED3 defines the ED sound as it occurs at the end of "heated".

In running the program, enter a 0 for the truncation value in order to leave the truncation sequence.

```
100 REM *******************
110 REM REQUIRES MERGE OF:
120 REM MENU (LINES 10000 THROUGH 10120)
130 REM DEFING (LINES 1000 THROUGH 1130)
140 REM DEFS1 (LINES 2000 THROUGH 2100)
150 REM DEFS2 (LINES 3000 THROUGH 3090)
160 REM DEFS3 (LINES 4000 THROUGH 4120)
170 REM DEFED1 (LINES 5000 THROUGH 5070)
180 REM DEFED2 (LINES 6000 THROUGH 6110)
190 REM DEFED3 (LINES 7000 THROUGH 7130)
200 REM *******************
210 CALL CLEAR
220 PRINT "THIS PROGRAM IS USED TO"
230 PRINT "FIND THE PROPER TRUNCATION"
240 PRINT "VALUE FOR ADDING SUFFIXES"
250 PRINT "TO SPEECH WORDS.": :
260 FOR DELAY=1 TO 300::NEXT DELAY
270 PRINT "CHOOSE WHICH SUFFIX YOU"
280 PRINT "WISH TO ADD.": :
```

```
290 FOR DELAY=1 TO 200::NEXT DELAY
300 CALL MENU(8,CHOICE)
310 DATA 'ING','S' AS IN CATS,'S' AS IN CADS,'S' AS IN WISHES,
'ED' AS IN PASSED,'ED' AS IN CAUSED,'ED' AS IN HEATED,END
320 IF CHOICE=0 OR CHOICE=8 THEN STOP
330 INPUT "WHAT IS THE WORD? ":WORD$
340 ON CHOICE GOTO 350,370,390,410,430,450,470
350 CALL DEFING(D$)
360 GOTO 480
370 CALL DEFS1(D$)!CATS
380 GOTO 480
390 CALL DEFS2(D$)!CADS
400 GOTO 480
410 CALL DEFS3(D$)!WISHES
420 GOTO 480
430 CALL DEFED1(D$)!PASSED
440 GOTO 480
450 CALL DEFED2(D$)!CAUSED
460 GOTO 480
470 CALL DEFED3(D$)!HEATED
480 REM TRY VALUES
490 CALL CLEAR
500 INPUT TRUNCATE HOW MANY BYTES? ":L
510 IF L=0 THEN 300
520 CALL SPGET(WORD$,B$)
530 L=LEN(B$)-L-3
540 C$=SEG$(B$,1,2)&CHR$(L)&SEG$(B$,4,L)
550 CALL SAY(,C$&D$)
560 GOTO 500
```

The data has been given in short DATA statements to make it as easy as possible to input. It may be consolidated to make the program shorter.

```
1000 SUB DEFING(A$)
1010 DATA 96,0,52,174,30,65
1020 DATA 21,186,90,247,122,214
1030 DATA 179,95,77,13,202,50
1040 DATA 153,120,117,57,40,248
1050 DATA 133,173,209,25,39,85
1060 DATA 225,54,75,167,29,77
1070 DATA 105,91,44,157,118,180
1080 DATA 169,97,161,117,218,25
1090 DATA 119,184,227,222,249,238,1
1100 RESTORE 1010
1110 A$=""
1120 FOR I=1 TO 55::READ
A::A$=A$&CHR$(A)::NEXT I
1130 SUBEND

2000 SUB DEFS1(A$)!CATS
2010 DATA 96,0,26
```

```
2020 DATA 14,56,130,204,0
2030 DATA 223,177,26,224,103
2040 DATA 85,3,252,106,106
2050 DATA 128,95,44,4,240
2060 DATA 35,11,2,126,16,121
2070 RESTORE 2010
2080 A$=""
2090 FOR I=1 TO 29::READ
A::A$=A$&CHR$(A)::NEXT I
2100 SUBEND

3000 SUB DEFS2(A$)!CADS
3010 DATA 96,0,17
3020 DATA 161,253,158,217
3030 DATA 168,213,198,86,0
3040 DATA 223,153,75,128,0
3050 DATA 95,139,62
3060 RESTORE 3010
3070 A$=""
3080 FOR I=1 TO 20::READ
A::A$=A$&CHR$(A)::NEXT I
3090 SUBEND

4000 SUB DEFS3(A$)!WISHES
4010 DATA 96,0,34
4020 DATA 173,233,33,84,12
4030 DATA 242,205,166,55,173
4040 DATA 93,222,68,197,188
4050 DATA 134,238,123,102
4060 DATA 163,86,27,59,1,124
4070 DATA 103,46,1,2,124,45
4080 DATA 138,129,7
4090 RESTORE 4010
4100 A$=""
4110 FOR I=1 TO 37::READ
A::A$=A$&CHR$(A)::NEXT I
4120 SUBEND

5000 SUB DEFED1(A$)!PASSED
5010 DATA 96,0,10
5020 DATA 0,224,128,37
5030 DATA 204,37,240,0,0,0
5040 RESTORE 5010
5050 A$=""
5060 FOR I=1 TO 13::READ A::A$=A$&CHR$(A)::NEXT I
5070 SUBEND

6000 SUB DEFED2(A$)!CAUSED
6010 DATA 96,0,26
6020 DATA 172,163,214,59,35
6030 DATA 109,170,174,68,21
```

```
6040 DATA 22,201,220,250,24
6050 DATA 69,148,162,166,234
6060 DATA 75,84,97,145,204
6070 DATA 15
6080 RESTORE 6010
6090 A$=""
6100 FOR I=1 TO 29::READ
A::A$=A$&CHR$(A)::NEXT I
6110 SUBEND

7000 SUB DEFED3(A$)!HEATED
7010 DATA 96,0,36
7020 DATA 173,233,33,84,12
7030 DATA 242,205,166,183
7040 DATA 172,163,214,59,35
7050 DATA 109,170,174,68,21
7060 DATA 22,201,92,250,24
7070 DATA 69,148,162,38,235
7080 DATA 75,84,97,145,204
7090 DATA 178,127
7100 RESTORE 7010
7110 A$=""
7120 FOR I=1 TO 39::READ A::A$=A$&CHR$(A)::NEXT I
7130 SUBEND

10000 SUB MENU(COUNT,CHOICE)
10010 CALL CLEAR
10020 IF COUNT>22 THEN PRINT "TOO MANY ITEMS" :: CHOICE=0 :: SUBEXIT
10030 RESTORE
10040 FOR I=1 TO COUNT
10050 READ TEMP$
10060 TEMP$=SEG$(TEMP$,1,25)
10070 DISPLAY AT(I,1):I;TEMP$
10080 NEXT I
10090 DISPLAY AT(I+1,1):"YOUR CHOICE: 1"
10100 ACCEPT AT(I+1,14)BEEP VALIDATE(DIGIT)SIZE(-2):CHOICE
10110 IF CHOICE<1 OR CHOICE>COUNT THEN 10100
10120 SUBEND
```

You can use the subprogram in any program once you have determined the number of bytes to truncate. The following program uses the subprogram DEFING in lines 1000 through 1130 to have the computer say the word DRAWING using DRAW plus the suffix ING. Note that it was found that DRAW should be truncated by 41 characters to produce the most natural sounding DRAWING. The subprogram DEFING in lines 1000 through 1130 is the program you saved with the merge option.

```
100 CALL DEFING(ING$)
110 CALL SPGET("DRAW",DRAW$)
120 L=LEN(DRAW$)-3-41! 3 BYTES OF SPEECH OVERHEAD, 41 BYTES TRUNCATED
130 DRAW$=SEG$(DRAW$,1,2)&CHR$(L)&SEG$(DRAW$,4,L)
140 CALL SAY("WE ARE",DRAW$&ING$,"A1 SCREEN")
```

```
150 GOTO 140
1000 SUB DEFING(A$)
1010 DATA 96,0,52,174,30,65
1020 DATA 21,186,90,247,122,214
1030 DATA 179,95,77,13,202,50
1040 DATA 153,120,117,57,40,248
1050 DATA 133,173,209,25,39,85
1060 DATA 225,54,75,167,29,77
1070 DATA 105,91,44,157,118,180
1080 DATA 169,97,161,117,218,25
1090 DATA 119,184,227,222,249,238,1
1100 RESTORE 1010
1110 A$=""
1120 FOR I=1 TO 55::READ A::A$=A$&CHR$(A)::NEXT I
1130 SUBEND
```

(Press **FCTN 4** to stop the program.)

Appendix J. Errors

The following lists all the error messages that MYARC Extended BASIC II gives. The first list is alphabetical by the message that is given, and the second list is numeric by the number of the error that is returned by CALL ERR. If the error occurs in the execution of a program, the error message is often followed by IN line-number.

Sorted by Message

*Message* *Descriptions of Possible Errors*

74 **BAD ARGUMENT**

* Bad value given in ASC, ATN, COS, EXP, INT, LOG, SIN, SOUND, SQR, TAN, or VAL.
* An array element specified in a SUB statement.
* Bad first parameter or too many parameters in LINK.

61 **BAD LINE NUMBER**

* Line number less than 1 or greater than 32767.
* Omitted line number.
* Line number outside the range 1 through 32767 produced by RES.

57 **BAD SUBSCRIPT**

* Use of too large or small subscript in an array.
* Incorrect subscript in DIM.

79 **BAD VALUE**

* Incorrect value given in AND, CHAR, CHR$, CLOSE, EOF, FOR, GOSUB, GOTO, HCHAR, INPUT, MOTION, NOT, OR, POS, PRINT, PRINT USING. REC, RESTORE, RPT$. SEG$. SIZE, VCHAR or XOR.
* Array subscript value greater than 32767.
* File number greater than 255 or less than zero.
* More than three tones and one noise generator specified in SOUND.
* A value passed to a subprogram is not acceptable in the subprogram. For example, a sprite velocity value less than - 128 or a character value greater than 143.
* Value in ON...GOTO or ON...GOSUB greater than the number of lines given.
* Incorrect position given after the AT clause in ACCEPT or DISPLAY.

67 **CAN'T CONTINUE**

* Program has been edited after being stopped by a breakpoint.
* Program was not stopped by a breakpoint.

69 **COMMAND ILLEGAL IN PROGRAM**

* BYE, CON, LIST, MERGE, NEW, NUM, OLD, RES, or SAVE used in a program.

84 **DATA ERROR**

* READ or RESTORE with data not present or with a string where a numeric value is expected.
* Line number after RESTORE is higher than the highest line number in the program.
* Error in object file in LOAD.

109 **FILE ERROR**

* Wrong type of data read with a READ statement.
* Attempt to use CLOSE, EOF, INPUT, OPEN, PRINT, PRINT USING, REC, or RESTORE with a file that does not exist or does not have the proper attributes.
* Not enough memory to use a file.

44 **FOR NEXT NESTING**

* The FOR and NEXT statements of loops do not align properly .
* Missing NEXT statement.

130 **I/O ERROR**

* An error was detected in trying to execute CLOSE, DELETE, LOAD, MERGE, OLD, OPEN, RUN, or SAVE.
* Not enough memory to list a program.

16 **ILLEGAL AFTER SUBPROGRAM**

* Anything but END, REM, or SUB after a SUBEND.

36 **IMAGE ERROR**

* An error was detected in the use of DISPLAY USING, IMAGE, or PRINT USING.
* More than 10 (E-format) or 14 (numeric format) significant digits in the format string.
* IMAGE string is longer than 254 characters.

28 **IMPROPERLY USED NAME**

- * An illegal variable name was used in CALL, DEF, or DIM.
- * Using a MYARC Extended BASIC II reserved word in LET.
- * Using a subscripted variable or a string variable in a FOR.
- * Using an array with the wrong number of dimensions.
- * Using a variable name differently than originally assigned. A variable can be only an array, a numeric or string variable, or a user defined function name.
- * Dimensioning an array twice.
- * Putting a user defined function name on the left of the equals sign in an assignment statement.
- * Using the same variable twice in the parameter list of a SUB statement.

81 **INCORRECT ARGUMENT LIST**

- * CALL and SUB mismatch of arguments.

83 **INPUT ERROR**

- * An error was detected in an INPUT.

60 **LINE NOT FOUND**

- * Incorrect line number found in BREAK, GOSUB, GOTO, ON ERROR, RUN, or UNBREAK, or after THEN or ELSE.
- * Line to be edited not found.

62 **LINE TOO LONG**

- * Line too long to be entered into a program.

39 **MEMORY FULL**

- * Program too large to execute one of the following: DEF, DELETE, DIM, GOSUB, LET, LOAD, ON...GOSUB. OPEN, or SUB.
- * Program too large to add a new line, insert a line, replace a line, or evaluate an expression.

49 **MISSING SUBEND**

- * SUBEND missing in a subprogram.

47 **MUST BE IN SUBPROGRAM**

- * SUBEND or SUBEXIT not in a subprogram.

19 **NAME TOO LONG**

- * More than 15 characters in variable or subprogram name.

43 **NEXT WITHOUT FOR**

- * FOR statement missing, NEXT before FOR, incorrect FOR-NEXT nesting, or branching into a FOR-NEXT loop.

78 **NO PROGRAM PRESENT**

* No program present when issuing a LIST, RESEQUENCE, RESTORE, RUN, or SAVE command.

10 **NUMERIC OVERFLOW**

* A number too large or too small resulting from a *, +, -, / operation or in ACCEPT, ATN, COS. EXP, INPUT. INT, LOG, SIN, SQR, TAN, or VAL.

* A number outside the range - 32768 to 32767 in PEEK or LOAD.

70 **ONLY LEGAL IN A PROGRAM**

* One of the following statements was used as a command: DEF, GOSUB, GOTO, IF, IMAGE, INPUT, ON BREAK, ON ERROR, ON...GOSUB, ON...GOTO, ON WARNING, OPTION BASE, RETURN, SUB, SUBEND, or SUBEXIT.

25 **OPTION BASE ERROR**

* OPTION BASE executed more than once, or with a value other than 1 or zero.

97 **PROTECTION VIOLATION**

* Attempt to save, list, or edit a protected program.

48 **RECURSIVE SUBPROGRAM CALL**

* Subprogram calls itself, directly or indirectly.

51 **RETURN WITHOUT GOSUB**

* RETURN without a GOSUB or an error handled by the previous execution of an ON ERROR statement.

56 **SPEECH STRING TOO LONG**

* Speech string returned by SPGET is longer than 255 characters.

40 **STACK OVERFLOW**

* Too many sets of parentheses.

* Not enough memory to evaluate an expression or assign a value.

54 **STRING TRUNCATED**

* A string created by RPT$, concatenation ("&" operator), or a user defined function is longer than 255 characters.

* The length of a string expression in the VALIDATE clause is greater than 254 characters.

24 **STRING NUMBER MISMATCH**

- * A string was given where a number was expected or vice versa in a MYARC Extended BASIC II supplied function or subprogram.
- * Assigning a string value to a numeric value or vice versa.
- * Attempting to concatenate ("&" operator) a number.
- * Using a string as a subscript.

135 **SUBPROGRAM NOT FOUND**

- * A subprogram called does not exist or an assembly language subprogram named in LINK has not been loaded.

14 **SYNTAX ERROR**

- * An error such as a missing or extra comma or parenthesis, parameters in the wrong order, missing parameters, missing keyword, misspelled keyword, keyword in the wrong order, or the like was detected in a MYARC Extended BASIC II command, statement, function, or subprogram.
- * DATA or IMAGE not first and only statement on a line.
- * Items after final ")".
- * Missing "#" in SPRITE.
- * Missing ENTER, tail comment symbol (!), or statement separator symbol (::).
- * Missing THEN after IF.
- * Missing TO after FOR.
- * Nothing after CALL, SUB, FOR, THEN, or ELSE.
- * Two E's in a numeric constant.
- * Wrong parameter list in a MYARC Extended BASIC II supplied subprogram.
- * Going into or out of a subprogram with GOTO, GOSUB, ON ERROR, etc.
- * Calling INIT without the Memory Expansion peripheral attached.
- * Calling LINK or LOAD without first calling INIT.
- * Using a constant where a variable is required.
- * More than seven dimensions in an array.

17 **UNMATCHED QUOTES**

- * Odd number of quotes in an input line.

20 **UNRECOGNIZED CHARACTER**

- * An unrecognized character such as ? or % is not in a quoted string.
- * A bad field in an object file accessed by LOAD.

Sorted by

#	*Message*
10	NUMERIC OVERFLOW
14	SYNTAX ERROR
16	ILLEGAL AFTER SUBPROGRAM
17	UNMATCHED QUOTES
19	NAME TOO LONG
20	UNRECOGNIZED CHARACTER
24	STRING NUMBER MISMATCH
25	OPTION BASE ERROR
28	IMPROPERLY USED NAME
36	IMAGE ERROR
39	MEMORY FULL
40	STACK OVERFLOW
43	NEXT WITHOUT FOR
44	FOR-NEXT NESTING
47	MUST BE IN SUBPROGRAM
48	RECURSIVE SUBPROGRAM CALL
49	MISSING SUBEND
51	RETURN WITHOUT GOSUB
54	STRING TRUNCATED
56	SPEECH STRING TOO LONG
57	BAD SUBSCRIPT
60	LINE NOT FOUND
61	BAD LINE NUMBER
62	LINE TOO LONG
67	CAN'T CONTINUE
69	COMMAND ILLEGAL IN PROGRAM
70	ONLY LEGAL IN A PROGRAM
74	BAD ARGUMENT
78	NO PROGRAM PRESENT
79	BAD VALUE
81	INCORRECT ARGUMENT LIST
83	INPUT ERROR
84	DATA ERROR
97	PROTECTION VIOLATION
109	FILE ERROR
130	I/O ERROR
135	SUBPROGRAM NOT FOUND

Appendix K. High-Resolution Mode

In order to stay as compatible as possible with TI Extended BASIC, certain restrictions are in affect when operating in the High-Resolution Mode. To insure proper execution of your program when programming in this mode, please refer to the restrictions and conventions listed below.

1. Only one floppy disk file may be kept open at any given time. If you wish to open a different floppy disk file you must first close any currently open floppy disk file. This restriction does not apply to other peripherals (RS232, RD, PIO, ... etc.).

2. Valid character codes extend from character ASCII 0 to 215. Characters 216 to 255 are not accessible.

3. Valid character sets accordingly are 0-23. Character sets 24-28 are not accessible.

4. Sprites function normally in High-Resolution Mode, withs the exception that sprite motion is no longer available. Attempts to put sprites into motion will have no effect.

5. ACCEPT and DISPLAY have no effect in High-Resolution Mode.

6. INPUT, LINPUT, PRINT, and PRINT USING can only be used in file access. Attempting to display characters on the screen with these commands/statements will have no effect.

7. WRITE, although somewhat limited, will allow you to display strings on the screen. You may also use HCHAR or VCHAR for screen displays in the High-Resolution Mode.

6. ADDENDUMS

6.1. Addendum 1. MYARC Extended BASIC Version 2.11

Your patience in waiting for V2.11 has been greatly appreciated. We feel the wait was worth it. In delivering to you such a fine product for the TI-99/4A. If you are an owner of V2.10 please use this addendum and disregard the old. Thank you for buying MYARC. We are working hard to keep your trust.

The following is an addendum of features covering V2.11 which are either not correct or are not covered in the manual. Following the addendum is a technical discussion on the architecture of V2.11 BASIC. Hopefully, this will, cover all details necessary to program efficiently in both BASIC and 9900 assembly language.

6.1.1. CLS

Typing in CLS will cause the display to be cleared. CLS is the equivalent of CALL CLEAR.

6.1.2. RUN "FILE-NAME", CONTINUE

Embedded in a program, this command allows you to load and execute a program, while maintaining the same "variable" values.

6.1.3. PWD — PRINT WORKING DIRECTORY, AND CHDIR — CHANGE WORKING DIRECTORY

Programmers have found working directories to be useful and convenient. Working directory pertains to OLD, SAVE, and RUN from the command-mode. When specifying a file, for any of these three commands, if a period is not specified file name, working directory is prefixed to the file name. Therefore, if working directory is DSK1. and you specify:

```
SAVE PROGRAM1
```

XBII would try to save program in memory to DSK1.PROGRAM1.

Upon power up, working directory is DSK1.. By issuing the CHDIR command, the working directory can be changed, to any name up to 15 characters.

6.1.4. OLD/SAVE/RUN

MYARC XBII uses VDP RAM more extensively (256 characters) than TI Extended Basic (114 characters). Therefore, when saving a Basic program, XBII will switch to Internal/Variable 254 format sooner than TI Extended Basic, because of less available VDP RAM.

In order to load all TI Extended Basic programs, in program image format, XBII may use the character definition table of VDP RAM as loadspace. The characters on the display will momentarily become undefined, but will be restored immediately after the program has been loaded. To prevent characters from being redefined in the future, perform a save after the program has been loaded.

MYARC XBII will automatically convert the format to Internal/Variable 254, usable by both TI Extended Basic and MYARC XBII. Because of the extra space available given by using the character definition table, MYARC XBII is able to load some TI Basic programs that TI Extended Basic was not able to load.

6.1.5. SIZE

The SIZE command will display the amount of program, variable, and string space available to the program.

6.1.6. FREESPACE

The FREESPACE command has been replaced by the SIZE command which provides more and better information covering memory utilization.

6.1.7. LIST

Please use the following commands in place of those in the manual:

LIST X-	Replaced by LIST X,E	or	LIST X E
LIST X-Y	Replaced by LIST X,Y	or	LIST X Y
LIST -X	Replaced by LIST 1,X	or	LIST 1 X

6.1.8. MARGINS

In High-Resolution Mode, margins have little value in BASIC and therefore are not supported in this mode.

Because XBII uses windows to present information, you may find a program or two that assume scrolling to occur in columns 1, 2 and 31 and 32. This will not happen. It works as TI defined it to operate on their own 9918.

6.1.9. REAL

Version 2.11 supports both INTEGER and REAL NUMBERS. However, the variable definition name for REAL types has been changed from REAL to DEFREAL.

6.1.10. RECTANGLE

Version 2.11 supports all modes of drawing pure vertical and horizontal shapes. In addition, general purpose parallelograms are supported in the TYPE 1 mode (i.e., drawing the perimeter).

6.1.11. FILL

Optional character pattern is not supported in V2.11.

6.1.12. GRAPHICS(3) MODE

When entering Graphics(3) mode, all available VDP RAM is needed, including some of the area normally used by the floppy disk controller. Therefore, upon invoking Graphics(3) mode, XBII will close all files and perform a "CALL FILES(1)". In this mode, only one floppy disk file can be open at a time. Another consideration must be given when using Graphics(3) mode, that of running other programs. Because all available VDP RAM is used for display, only loading of Internal/Variable 254 program files are allowed. In order to save a program in Internal/Variable 254 format, just issue the command:

```
SAVE "DSKX.PGMNAME",INTERNAL.
```

6.1.13. ! P+, ! P-, ! P*

TI Extended Basic supports the commands ! P+ and ! P- to speed up program prescan. MYARC XBII also supports these and has an additional command of ! P*. This new command stops prescan altogether. It differs from ! P- in that ! P- allows syntax checking to continue, whereas with ! P*, it does not. Because the new command stops prescan entirely at that point, it speeds program start-up time considerably. At the same token, EXTREME CARE must be taken when using this capability.

6.1.14. CALL SAY

When using CALL SAY, the computer will spell any alpha word string that is not in the library and say "UHOH" if the word string is non alphanumeric.

6.1.15. DEF

The DEF function can only support one and not multiple parameters.

6.1.16. RUN and OLD

The RUN and OLD command's cannot accept D/V80 format files. These are enhancements for the new MYARC Advanced BASIC for the TI-99/4A compatible 9640 computer.

6.1.17. Extended BASIC II RAM Usage

The following three diagrams depict memory usage of both CPU and VDP RAM by XBII.

6.1.17.1. EXTENDED BASIC II CRAM MAP

	I	II	III	IV
>2000 >4000	UNUSED ------------- USER ASSM LANG AREA	RAM DISK ------------- BASIC INTRP	VDP/SPEECH ROUTINES ------------- I/O BUFFERS	BASIC INTRP ------------- VALUE STACK
>6000 >8000	MEMORY MANAGEMENT ROUTINES			
>A000 >FFE8	BASIC PROGRAM	VARIABLE SPACE	STRING SPACE	BASIC INTRP

6.1.17.2. EXTENDED BASIC II VRAM MAP

PATTERN MODE	TEXT MODE	BITMAP MODE
>0 SIT 0 >2FF 767 ------------------------- >300 SAL 768 >37F 895 ------------------------- >380 CT 896 >39F 927 ------------------------- >3A0 EMPTY 928 >7FF 2047 ------------------------- >800 PDT/SDT 2048 >FFF 4095 ------------------------- >1000 EMPTY 4096 >37D7 14295 ------------------------- >37D8 RESERVED 14196 BLOCKS FOR FLOPPY DSR 3 FILES NORMAL PWR-UP MODE >3FFF	>0 SIT 0 >3BF 959 ------------------------- >3C0 EMPTY 960 >7FF 2047 ------------------------- >800 PDT 2048 >FFF 4095 ------------------------- >1000 EMPTY 4096 >37D7 14295 ------------------------- >37D8 RESERVED 14196 BLOCKS FOR FLOPPY DSR 3 FILES NORMAL PWR-UP MODE >3FFF	>0 PDT 0 >17FF 6143 ------------------------- >1800 SDT 6144 >1EBF 216 CHARs 7871 ------------------------- >1EC0 SOUND 7872 >1EC8 LIST 7880 ------------------------- >1EC9 311 BYTES 7881 FOR PAB, FILE DESCRIPTOR >1FFF 305 NEEDED 8191 ------------------------- >2000 CT 8192 >37FF ------------------------- >3800 SIT >3AFF ------------------------- >3B00 SAL >3AFF ------------------------- >3B80 EMPTY >3BE3 ------------------------- >3BE4 BLOCKS FOR DSR >3FFF 1 FILE

6.1.17.3. EXTENDED BASIC II VREG MAP

VDP REGS	VREG 0	VREG 1	VREG 2	VREG 3	VREG 4	VREG 5	VREG 6	VREG 7	SPR MOTN
GRAPHICS	MODE	MODE	SIT	CT	PDT	SAL	SDT	S/T	SMT (@>7A5C)
PATTERN	>00	>E0	>00 >0000	>0E >0380	>01 >0800	>06 >0300	>01 >0800	>17 BLK/ CY	xxxxx >7A5C
TEXT	>00	>F0	>00 >0000	>0F >03C0	>01 >0800	>xx >xxxx	>xx >xxxx	>17 BLK/ CY	---- ----
BITMAP	>02	>E0	>0E >3800	>FF >2000	>03 >0000	>76 >3B00	>03 >1800	>17 BLK/ CY	xxxxx >7A5C

6.1.18. ASSEMBLY LANGUAGE USAGE

As the CPU RAM usage depicts, a little over 1K-bytes of RAM is wasted in order to load assembly language programs at the exact same locations as TI XB. In order to use this extra area, immediately perform a CALL LOAD(8194,32,130) after your CALL INIT.

6.1.19. ASSEMBLY LANGUAGE SUPPORT

All assembly language support routines listed on pages 415-416 of the Editor/Assembler manual are supported except for the routines dealing with manipulation of data in VDP RAM.

These routines are next, COMPCT, GETSTR, MEMCHK, VPUSH, VPOP, ASSGNV, VGWITE, GVWITE. If these routines are invoked using an XML, they will return as a NO-OP. The reason for not allowing the user to invoke these routines (even though they are within XBII) is because no VDP RAM contains no data to be manipulated. All data is stored in CPU RAM, and therefore there is no need for these routines. A description of these routines is given in Chapter 17 in the E/A manual.

In addition, the assembly language loader in XBII is similar to that of the E/A Loader, in that compressed as well as non-compressed object code can be loaded. The following symbols are also predefined and can be used in your software by using the REF directive: PAD, GPLWS, SOUND, VDPRD, VDPSTA, VDPWD, VDPWA, SPCHRD, SPCHWT, GRMRD, GRMRA, GRMWD, GRMWA, SCAN, XMLLNK, KSCAN, VSBW, VSBR, VMBW, VMBR, VWTR, DSRLNK, LOADER, NUMASG, NUMREF, STRASG, STRREF.

As another enhancement, as the XBII manual states on page 136, when a "TYPE 5" assembly language program is used in an OLD or RUN statement, the program is loaded and executed immediately. Lastly, V2.11 supports all the returns given in section 24.11 in the E/A manual. However, if the alternate return as given in section 24.11.3 is used, the word in memory location 1/2-8300 should not be altered, as it contains a return linkage to XBII.

6.1.20. UTILITIES AND DEMONSTRATION PROGRAM

In addition, two new utilities and a demonstration program have been added.

The first utility is file name "128KOSN". The file differs from 128KOS in that it attempts to determine if XBII files 1-6 have already been loaded and therefore will only load "XBII7". This means that if you keep power on the RAM DISK and keep it in the 128K CPU RAM mode, you only need to load "XBII7" once. You can go into and out of XBlI from then on by only loading XBII7. This of course makes load time almost seven times faster. To use this capability you must rename file "128KOS" to something like "128KOSOLD" and rename "128KOSN" to "128KOS". You are now set.

The second utility is "TIVDP". This utility has been developed to solve some of the problems involving compatibility with loading assembly language from XB. The utility causes VDP RAM and VDP registers to be the same as used in TI XB. To invoke this utility a program statement should be as follows:

```
CALL INIT::CALL LOAD("DSK1.TIVDP")::CALL INIT
```

This will cause the screen to go blank and therefore should be used in a running program. To return VDP RAM and registers to XBII mode, simply type "NEW" and **ENTER**.

The demo program is called DEMO3M and shows some of the capabilities of XBII. To run it, type

```
RUN DSK1.DEMO3M.
```

6.2. Addendum 2. EPROM Installation Instructions

DEAR VALUED CUSTOMER:

ENCLOSED FIND YOUR UPDATED EPROM REPLACEMENT, AS REQUESTED. PLEASE RETURN THE OUTDATED EPROM TO US, ASAP. AS YOU KNOW, OUR CURRENT COMPANY POLICY IS TO SUPPLY UPDATED EPROMS AT NO CHARGE. YOUR CO-OPERATION IN RETURNING EPROMS, WILL ENABLE US TO CONTINUE TO SUPPLY SUBSEQUENT EPROM UPDATES, WITHOUT CHANGE IN OUR COMPANY POLICY.

1. Shutdown the console, PEB, and all other peripherals connected to your system.

2. Wait at least 2 minutes for power discharge, remove the PEB cover and lift out your MYARC card from the PEB.

3. The card is opened by separating the two "clamshell" halves at the edge OPPOSITE to the edge with the MYARC label. Note the two plastic catches at BOTH ends of this separation edge which lock the two clamshells together. You will need to depress BOTH catches, first at one end and then at the other end of the separation edge.

   The separation is best done using a medium size screwdriver (1/4 to 3/8" wide) to depress each catch INWARD while simultaneously pushing apart the two clamshells at that end. At each end, start with the outer catch first.

   After separation, lay the two halves on a table. The back side or the circuit board will be exposed.

   The two clamshells are NOT identical — note which clamshell holds the circuit board and the orientation of the board in that clamshell.

4. Holding it by the edges, lift out the circuit board and place it on a protected surface, back-side DOWN and card-edge connector contacts TOWARDS you.

6.2.1. REMOVING EPROM FROM CIRCUIT BOARD

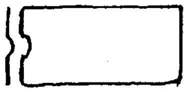
5. The EPROM is a large 28-pln IC and is located in the bottom right-hand quadrant of the board (nearest you) at location marked U_. The EPROM is further recognized by a small label affixed over and covering the window in the center of the EPROM.

*Note that the notch on the EPROM lines up with the drawing of a notch on the circuit board next to it.

6. To avoid bent or broken pins etc., extreme care must be used in removing the EPROM from its socket. Unless you're using a 28-pln IC remover, insert a small screwdriver end (up to 1/4" wide) first at one end, between the EPROM and the socket and GENTLY pry up that end of the EPROM a slight amount. At the other end, similarly place the screwdriver between the EPROM and the socket and then gently prying up that end about the some amount. Repeat the process alternately at each end so that the EPROM lifts up uniformly and easily from its socket.

6.2.2. INSERTING THE REPLACEMENT EPROM

7. After removing the new EPROM from its packing, orient it in the same direction that the old EPROM had been socketed.



WARNING: The EPROM will be irreparably damaged if inserted in the wrong direction and powered up.

8. Align all 28 pins of the EPROM into the 28 (rectangular) holes of the socket in the board. To get good alignment it may be necessary to adjust the EPROM pins, generally by carefully pressing them inward (towards the center) or otherwise as required, so that all 28 pins are well aligned to slide easily into the socket holes. Before pressing down, check again that all 28 pins are properly aligned over the 28 socket holes.

6.2.3. REPLACING THE CIRCUIT BOARD INTO THE CLAMSHELLS

10. Reversing the removal procedure, replace the circuit board, back-side up, into the proper clamshell. Adjust the circuit board so that two holes in the board fit over and into, the two supports protruding up from the clamshell.

11. Orient the other clamshell over the board. The card edge opening in the upper clamshell must be over the card edge connector of the circuit board and the small opening in the upper clamshell for the LED (lamp) must align with the LED on the circuit board.

12. Interlock the two small plastic hinges at the label-edge of the clamshells and then firmly push together each end at the separator edge. The plastic catches should snap into place with applied firm pressure.

13. Reinsert your MYARC card into your PEB and replace the PEB cover before system startup.

14. Please replace the old EPROM into its packing material and return it to MYARC, Inc., P.O. Box 140, Basking Ridge, NJ 07920.

6.2.4. NOTICE

THE SOFTWARE CONTAINED IN THE MYARC EPROM IS COPYRIGHTED BY MYARC AND MAY NOT BE COPIED OR DUPLICATED IN WHOLE OR IN PART FOR ANY REASON WHATSOEVER.

ATTEMPTS TO COPY OR TAMPER WITH THE MYARC EPROM ELECTRONICALLY OR OTHERWISE WILL PERMANENTLY DAMAGE THE EPROM.

EPROMs THAT HAVE BEEN DAMAGED DUE TO SUCH TAMPERING ARE NOT COVERED BY WARRANTY AND WILL NOT BE REPLACED BY MYARC.

7. SERVICE INFORMATION

7.1. In Case of Difficulty

If MYARC Extended BASIC II does not appear to be working properly, check the following:

1. Did you insert a high-quality Memory Expansion Card with a minimum of 128 Kbytes of RAM storage into your peripheral expansion box, AND did you ALSO REMOVE the 32K Memory Expansion Card from the PEB?

   To use MYARC Extended BASIC II in your 99/4A:

   > You MUST have 128 Kbytes minimum of memory expansion.
   >
   > AND
   >
   > You MUST NOT have a 32K Memory Expansion Card in the PEB together with the larger capacity Memory Expansion Card.

2. Power — Be sure all devices are plugged in. Then turn on the power to the units in the proper sequence: Peripheral devices first (if you have them), followed by the console and monitor.

3. Connector Separation — Check for proper alignment of the console and any accessory devices such as the Disk Drive Controller, Speech Synthesizer, and RS232 Interface. Remove and reinsert the MYARC Extended BASIC II module.

4. If none of the above procedures corrects the difficulty, consult "If You Have Questions or Need Assistance" or see the "Service Information" portion of the *User's Reference Guide* that came with your computer.

7.2. If You Have Questions or Need Assistance

If you have questions concerning MYARC Extended BASIC II operation or repair, contact first the dealer from whom you purchased the equipment.

Your dealer will be able to quickly answer most questions. If your dealer doesn't have an immediate answer, the dealer will either contact MYARC or suggest you contact us directly by mail or phone.

Our address and telephone number are:

MYARC, INC.
P. 0. BOX 140
Basking Ridge, NJ 07920
(201)766-1701

Please Note: This telephone number is not a toll-free number and collect calls cannot be accepted.

8. 90-DAY LIMITED WARRANTY

THIS MYARC EXTENDED BASIC II WARRANTY EXTENDS TO THE ORIGINAL CONSUMER PURCHASER OF THE ACCESSORY.

8.1. Warranty Duration

This MYARC Extended BASIC II module is warranted for a period of 90 days from the date of the original purchase by the consumer.

8.2. Warranty Coverage

This MYARC Extended BASIC II module is warranted against defective materials or workmanship. THIS WARRANTY IS VOID IF THE ACCESSORY HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.

8.3. Warranty Disclaimers

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE 90-DAY PERIOD. MYARC SHALL NOT BE LIABLE FOR LOSS OF USE OF THE HARDWARE OR OTHER INCIDENTAL OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER.

Some states do not allow the exclusion or limitation of implied warranties or consequential damages, so the above limitations or exclusions may not apply to you in those states.

8.4. Legal Remedies

This warranty gives you specific legal rights, and you may also have other rights that vary from state to state.

8.5. Warranty Performance

During the above 90-day warranty period, your MYARC Extended BASIC II module will be repaired or replaced with a new or reconditioned unit of the same or equivalent model (at MYARC's option) when return is authorized by MYARC and the unit is returned by prepaid shipment to MYARC, INC. at the address shown below. The repaired or replacement unit will be warranted for 90 days from date of repair or replacement.

Other than the shipping requirement, no charge will be made for the repair or replacement of in-warranty units.

SHIPPING INSTRUCTIONS: If you believe that your unit requires servicing, please contact MYARC before you return your unit. We will try to analyse and may be able to solve your problem without need of returning the unit. Please obtain a Return Authorization number from us before you ship the unit back.

MYARC strongly recommends that you insure the unit for value, prior to shipment.

MYARC's ADDRESS:

MYARC, INC.
241 Madisonville Road
Basking Ridge, NJ 07920

Statement of File Origin

This file was created for users of PC99, a TI-99/4A emulator running on an IBM PC.

> "Lou Phillips, former chief executive officer of Myarc Incorporated of Basking Ridge, NJ has granted the PC99 developers permission to reproduce the manual originally supplied with the MYARC Extended BASIC II package. The developers of PC99 acknowledge their thanks for this generous offer."

While every effort was made to ensure that the text and graphics content of this file are an accurate copy of the original Myarc manual, CaDD Electronics can assume no responsibility for any errors introduced during scanning, editing, or conversion.

If you find an error, we will attempt to correct it and provide you with an updated file. You can contact us at:

CaDD Electronics
45 Centerville Drive
Salem, NH 03079-2674

Version 971030