VBASICA

Rev. b

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

First VICTOR printing April, 1985.

ISBN 0-88182-137-3 Printed in U.S.A.

4

VBASICA Changes

This section documents features of VICTOR's VBASICA that are enhanced relative to IBM's BASICA, lists features that are not supported in this implementation of VBASICA, and discusses some minor compatibility problems and their resolutions.

4.1 Enhanced Features

Features enhanced over BASICA include the LCOPY statement, the VBCONF program, bit map allocation, and additional graphics screens and modes.

4.1.1 LCOPY Statement

FORMAT:

```
LCOPY
```

PURPOSE:

Dumps the screen display to a graphics printer.

REMARKS:

With the LCOPY statement VBASICA can output both text and graphics to a dot-matrix printer. VBASICA supports seven printers; you can also define additional printers. Before you can use LCOPY, you must install VBASICA for the printer you are using. Do so by running the VBCONF.BAS program provided on your distribution diskette (VBCONF is described in the next section). After you install VBASICA for a particular printer, you can use the LCOPY statement only with that printer.

EXAMPLE:

```
10 SCREEN 2                  'set hi-res graphics screen
20 KEY OFF                   'turn off function key display
30 CLS 2                     'clear the screen
40 FOR RADIUS = 20 TO 200 STEP 10
50 CIRCLE (400,200),RADIUS     'draw some graphics
60 NEXT RADIUS
70 LCOPY                     'copy the screen to the printer
80 END
```

4.1.2 The VBCONF Program

You can run the VBCONF configuration program either from the operating system level or from VBASICA. To run VBCONF from the operating system level, enter the following command in response to the system prompt:

```
VBASICA VBCONF
```

To run VBCONF from the VBASICA command level, enter this command in response to the "Ok" prompt:

```
RUN "VBCONF"
```

The program displays the following menu choices:

ABORT - Abort this program without changing VBASICA.
VICTOR Printer 6010/6020
VICTOR Printer 6015/6025
VICTOR Printer 6070/6075
Tally Printer (160s/180s)
Tally Printer (140)
C. Itoh Printer (8510A, 8510S/1550A, 1550S)
Epson MX-80/MX-100 (with GRAFTRAX)
Epson FX-80/FX-100
Okidata Printer (U84 Only)
No Printer - disable LCOPY

Use the cursor arrow keys to move the reverse-video bar up and down the menu. When the choice you want is highlighted, press Return. The V9BASICA.XEC file on the default drive is modified and you return to the operating system. If you choose to ABORT, you return to VBASICA. If you choose "No Printer," LCOPY returns an "Illegal function call" error message if you try to use it.

The VBASICA sign-on message displays the name of the installed printer. Rerun VBCONF any time you change printers.

You can install a printer not listed in the menu by modifying the VBCONF.BAS source code. After reading your printer's manual and the comments in the VBCONF.BAS source code, you should be able to configure VBASICA for most printers that support bit-mapped dot-matrix printing.

VICTOR dot-matrix printers automatically correct the aspect ratio between the printer and the screen so that circles are printed correctly, rather than slightly elongated, as is the case with most printers.

4.1.3 Bit Map Allocation

The graphics functions of VBASICA require a bit map. A **bit map** is memory that defines the graphics screen; this memory is loaded in the lower 64K of RAM.

The bit map for VBASICA is usually allocated at run time by the VBASICA loader. Many load-and-stay-resident utilities, however, load in low memory, thus preventing the VBASICA loader from allocating the bit map. To overcome this conflict, two utilities—GETSCRN and KILLSCRN—have been provided to allocate and deallocate the memory for the bit map. You can run these utilities at any time.

To avoid conflicts with resident utilities, first run GETSCRN to reserve memory for the bit map, and then load your other utilities. You can then run VBASICA at any time, and it will be able to use the bit map allocated by GETSCRN. If you run GETSCRN before VBASICA, VBASICA knows that memory for the bit map has already been allo-

cated and does not attempt to allocate more memory. You can return the reserved memory to the system by using KILLSCRN.

Communication between GETSCRN and VBASICA is accomplished as follows. GETSCRN allocates the bit map (if it can), stores the segment address of the bit map in the offset of interrupt D9 (location 0:364), and stores a "BM" (4D42) in the segment of interrupt D9 (location 0:366) to indicate a successful allocation. GETSCRN and VBASICA check interrupt D9 with each invocation; if the bit map has already been allocated, they do not allocate another one.

4.1.4 Graphics Screens and Modes

VBASICA can create medium- or high-resolution graphics on a color screen or on a standard screen. If your computer has a color card, you can put color and black-and-white images on an attached color screen. VBASICA supports three screen types:

- The standard VICTOR screen
- A software-simulated IBM color screen displayed on the standard VICTOR screen
- The color screen, which requires a color screen and the VICTOR IBM-compatible color card

In addition to three screen types, VBASICA supports three screen resolution modes:

- Mode 0: text-only mode
- Mode 1: medium-resolution graphics and text mode
- Mode 2: high-resolution graphics and text mode

You can set both the screen type and the screen mode from within a VBASICA program by using the SCREEN command. Nine screen/mode combinations are possible, although only eight are implemented.

The possible combinations are shown in Table 1. Each screen number in the table is a two-digit number. The first digit specifies the screen type; the second digit specifies the resolution mode. Use these screen numbers as arguments to the SCREEN command to switch from one type/mode combination to another.

Table 1: Screen Type and Mode Combinations

Screen	Rows	Graphics Columns	Pixels	Comments
Color Screens				
20	25	40/80	No graphics	Full 16 colors, 8 or 4 pages, depending on width
21	25	40	320 × 200	4 colors, medium resolution
22	25	80	640 × 200	2 colors: black and white
Standard Screens				
40	25	40/80	No graphics	Standard text mode
41	(Not available; use screen 61)			
42	25	80	400 × 800	Standard high-resolution screen
IBM Monochrome Screens				
60	25	40/80	No graphics	IBM text mode
61	25	40	320 × 200	4 shades, medium resolution
62	25	40/80	640 × 200	Simulated IBM high-resolution monochrome mode

When the operating system loads VBASICA, it checks to see whether or not a color card is installed and whether or not the bit map is allocated.

If a color card is installed, the VBASICA default screen is screen 20. If no color card is installed, the VBASICA default screen is screen 40.

If the bit map is not allocated, your programs cannot access graphics screens 42, 61, and 62.

After you set a screen type/mode combination using one of the two-digit screen numbers shown in Table 1, you can keep the same screen type but change the resolution mode by issuing a SCREEN command followed by the one-digit mode number (this procedure corresponds to the SCREEN command description in your VBASICA manual).

The following command, for example, switches to the standard screen in text mode:

```
SCREEN 40
```

You can then use the following command to switch to the standard screen in high-resolution mode:

```
SCREEN 2
```

If you want to run color graphics programs but do not have a color screen, issue the command SCREEN 60 to simulate the IBM color screen on the standard screen.

Color Attributes

You can specify a color attribute with the graphics statements PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW. The range is 0 to 3. These color attribute numbers are distinct from the numbers that refer to actual colors; the latter are used only as parameters in the COLOR statement.

On screen 61, 0 selects black; 1, 2, and 3 select varying shades of white.

In Mode 1 on the color screen (screen 21), 0 selects the background color; 1, 2, and 3 select foreground colors.

In Mode 2 (screen 22 or screen 62), 0 and 2 select black; 1 and 3 select white.

Note: The COLOR statement does not affect any graphics screen except screen 21.

Coordinates

The drawing statements PSET, PRESET, LINE, CIRCLE, GET, PUT, and PAINT require you to specify points on the screen as pairs of x-y coordinates. Specify coordinates in the format (x, y), where **x** and **y** are numeric expressions. The ranges of x and y values for each screen type/mode combination are shown in Table 2.

Table 2: Screen Coordinates

Screen Type/Mode	X-Range (Horizontal)	Y-Range (Vertical)
21, 61	0–319	0–199
42 (standard screen)	0–799	0–399
22, 62 (color screen)	0–639	0–199

The upper left corner of the screen is always point (0, 0).

When you clear the screen with either the SCREEN statement or the CLS statement, the graphics cursor is set to the middle of the screen. Table 3 gives the midscreen coordinates for each screen type/mode combination.

Table 3: Midscreen Coordinates

Screen	Coordinates
21, 61	(160, 100)
42	(400, 200)
22, 62	(320, 100)

4.2 Features Not Supported in This Version of VBASICA

The following features are not supported:

- The PAINT statement does not support tiling.
- The LINE statement does not support the style attribute.
- The PLAY statement does not support the incrementing and decrementing octaves option.
- User-defined trappable keys are not supported. Only keys 0 through 11 can be trapped.
- The MS-DOS PATH command and the use of pathnames in file specifiers are not supported.
- I/O redirection is not supported.

These statements, commands, and functions are not supported:

CHDIR command
ENVIRON statement
ENVIRON$ function
ERDEV function
ERDEV$ function
IOCTL statement
IOCTL$ function
MKDIR command
ON PLAY statement
ON TIMER statement
PLAY OFF statement
PLAY ON statement
PLAY STOP statement
PLAY (n) function
PMAP function
RANDOMIZE statement
RMDIR command
SHELL statement
TIMER OFF statement
TIMER ON statement
TIMER STOP statement
VIEW statement
VIEW PRINT statement
WINDOW statement

4.3 Compatibility Problems and Resolutions

VBASICA is compatible with IBM BASICA, and programs written to run under IBM BASICA should run under VBASICA. There are, however, three differences between VBASICA and BASICA that you should be aware of.

4.3.1 Screen Compatibility

VBASICA supports three screen types, while BASICA supports one type. If you have a color screen, the difference in the screen types should not affect the operation of the BASICA program because the color screen is IBM compatible. VBASICA defaults to screen 20 (color screen, text mode) if you have a color card installed. Thus, as long as the program switches resolution modes when needed (SCREEN 0, 1, or 2), the program should run normally.

If you do not have a color screen, VBASICA defaults to screen 40 (VICTOR screen, text mode). Screen 40 is **not** an IBM compatible screen. Therefore, before the BASICA program accesses the screen, the screen mode should be changed to Screen 60 (IBM simulated screen, text mode). The easiest way to do this is to add the statement SCREEN 60 to the BASICA program so that it is the first statement executed. Then, as long as the program switches resolution modes when needed (SCREEN 0, 1, or 2), it should run normally.

4.3.2 Hardware Compatibility

The second area of difference is in the hardware. If a program attempts to access directly an IBM hardware feature that is unavailable on the VICTOR 9000, the program will not run correctly under VBASICA. Known areas of hardware incompatibility are as follows:

- BLOADs of a binary file from a hard disk to the color card do not work. Only part of the file is loaded. The IBM-compatible color card does not work correctly with the hard disk controller because the controller attempts to write to the color card's memory using Direct Memory Access (DMA).

  You can overcome this problem by first reading the contents of the binary file into an array and then using POKE to move the contents of the array into the color card's memory. The color card memory begins at &hB800:0.

  BLOADs from diskette to the color card work correctly because they do not use DMA.

- For some statements that include a boolean (logical true or false) value or variable, the value used for true must be greater than zero but less than 256. For example, in the LOCATE statement

  ```
  LOCATE LIN%, COL%, CRSR.ATTR%
  ```

  the variable CRSR.ATTR% is a boolean value. If CRSR.ATTR% is true, the cursor is on; if it is false, the cursor is off. If CRSR.ATTR% is true, it must be in the range

  $1 <= \text{CRSR.ATTR\%} <= 255$

  CRSR.ATTR% cannot be negative. To avoid problems, use + 1 for true and 0 for false.

4.3.3 Function Keys

VBASICA supports a maximum of seven function keys. Therefore, if you have a BASICA program that uses function keys F8 through F10, you will have to rewrite the program to use function keys F1 through F7 only.

VBASICA

Rev. C

VICTOR Publications
380 El Pueblo Road
Scotts Valley, California 95066
(408) 438-6680

First VICTOR printing August, 1985.

ISBN 0-88182-148-9 Printed in U.S.A.

Important Software Diskette Information

For your own protection, do not use this product until you have made a backup copy of your software diskette(s). The backup procedure is described in the user's guide for your computer.

Please read the DISKID file on your new software diskette. DISKID contains important information including:

- The part number of the diskette assembly.
- The software library disk number (for internal use only).
- The date of the DISKID file.
- A list of files on the diskette, with version number, date, and description for each one.
- Configuration information (when applicable).
- Notes giving special instructions for using the product.
- Information not contained in the current manual, including updates, any known bugs, additions, and deletions.

To read the DISKID file onscreen, follow these steps:

1. Load the operating system.
2. Remove your system diskette and insert your new software diskette.
3. Enter —

   type diskid(cr)

4. The contents of the DISKID file is displayed on the screen. If the file is large (more than 24 lines), the screen display will scroll. Type CTRL-S to freeze the screen display; type CTRL-S again to continue scrolling.

Preface

VICTOR BASIC Advanced (VBASICA) is an IBM PC-compatible extension to MS-BASIC. This manual includes MS-BASIC commands and the advanced commands.

VBASICA takes advantage of the facilities of the newer 16-bit microprocessors. The extra capabilities provide:

- Advanced graphics
- Sound
- Disk I/O and telecommunications support
- Event trapping

Using This Reference Manual

Chapter 1 gets you started with VBASICA. Chapter 1 also covers device-independent I/O and files; and both initialization and printer configuration are described.

Chapter 2 discusses the character set and explains how to edit your programs with the full screen editor. This editor provides immediate visual feedback and speeds editing functions such as cursor movement, insertion, and deletion. Chapter 2 also describes how VBASICA handles different data types.

Chapter 3 describes all the VBASICA statements, commands, and functions.

Chapter 4 describes the communications option, including Communication I/O, the TTY program, and COM I/O functions.

There are six appendixes and an index. Appendix A describes error messages. Appendix B lists key codes. Appendix C is an ASCII character code table. Appendix D includes the math functions. Appendix E describes how to convert programs to VBASICA. Appendix F discusses VBASICA disk I/O.

Manual Conventions

This manual uses the following conventions:

- The parts of the command line that you type are shown in uppercase. You can type the command in uppercase or lowercase. For example:

```
A>COPY WORK.TXT B:PAYROLL.DOC
```

- Parts of the command line that are optional or variable are shown in lowercase. Square brackets indicate optional portions of the command.
- A (cr) in the command line indicates that you press the Return key.
- In the text, names of commands, files, and programs appear in all-uppercase—for example, COPY or MEMO.TXT. You can enter the names in uppercase or lowercase.
- In the text examples, the system prompt is assumed to be A>, unless otherwise specified. Your system prompt might have additional characters.

Contents

TABLES

Chapters

1

Getting Started

1.1 Introduction

This chapter describes VBASICA's special features, such as graphics, device-independent I/O, and event trapping.

1.2 Modes of Operation

When VBASICA is initialized, it types the prompt "Ok." "Ok" indicates that VBASICA is at command level—that is, it is ready to accept commands. At this point, you can use VBASICA in either of two modes: the direct mode or the indirect mode.

In the direct mode, line numbers do not precede VBASICA statements and commands. VBASICA executes them as they are entered. Results of arithmetic and logical operations can be displayed immediately and stored for later use, but the instructions are lost after execution. This mode is useful for debugging and for using VBASICA as a calculator for quick computations that do not require a complete program.

Use the indirect mode for entering programs. VBASICA precedes program lines with line numbers and stores them in memory. You execute the program in memory by entering the RUN command.

1.3 Line Format and Line Numbers

Program lines in a VBASICA program have the following format, with square brackets indicating optional information:

nnnn < VBASICA statement > [: < VBASICA statement > ...](cr)

where *nnnn* is the line number. You can place more than one VBASICA statement on a line, but each statement on a line must be separated from the last by a colon. A VBASICA program line always begins with a line number, ends with a Return, and can contain up to 255 characters.

1.4 Initialization

FORMAT:

BASICA [< filename >] [/F: < number of files >] [/S: < lrecl >]
[/C: < buffer size >] [/M: < highest memory location >]

REMARKS:

Load and run VBASICA by typing the following command at the MS-DOS command line prompt:

basica(cr)

Upon loading, VBASICA responds with its banner.

You can alter the VBASICA operating environment somewhat by specifying option switches following VBASICA on the command line; these switches are described in the next paragraphs.

< filename > The filename of a VBASICA program. Files and filenaming conventions are described in the next section. If < filename > is present, VBASICA proceeds as if you gave a RUN " < filename > " command after initialization. VBASICA assumes a default file extension of .BAS if none is given. If you use this form of the command line in VBASICA programs, you can run the programs in batch files. Such programs must exit with the SYSTEM command so VBASICA can execute the next command in the batch file.

/F: < number of files > If present, sets the maximum number of files that can be open simultaneously during the execution of a VBASICA program.

Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size can be altered via the /S: option switch. If the /F: option is omitted, the number of files defaults to 3.

/S: < lrecl > If present, sets the maximum record size allowed for use with random files. **Note:** The record size option to the OPEN statement cannot exceed this value. If you omit the /S: option, the record size defaults to 128 bytes.

/C: < buffer size > If present, controls RS-232-C communications. If /C:0, RS-232-C support is disabled. Any subsequent I/O attempts result in the "Device Unavailable" error. Specifying /C: < n > allocates < n > bytes for the receive buffer and 128 bytes for the transmit buffer for each port. If you omit the /C: option, 256 bytes are allocated for the receive buffer and 128 bytes for the transmit buffer of each card present.

/M:<highest memory location> When present, sets the highest memory location that VBASICA uses. VBASICA attempts to allocate 65K of memory for the data and stack segments. To use machine language subroutines with VBASICA programs, use the /M: switch to reserve enough memory for them.

1

NOTE: <number of files>, <lrecl>, <buffer size>, and <highest memory location> can be decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

EXAMPLE:

Use all memory and 3 files; load and execute PAYROLL.BAS:

```
A>BASICA PAYROLL
```

Use all memory and 6 files; load and execute INVENT.BAS:

```
A>BASICA INVENT/F:6
```

Disable RS-232-C support and use only the first 32K of memory:

```
A>BASICA /C:0/M:32768
```

Use 4 files and allow a maximum record length of 512 bytes:

```
A>BASICA /F:4/S:512
```

Use all memory and 3 files; allocate 512 bytes to RS-232-C receive buffers and 128 bytes to transmit buffers, load and execute TTY.BAS:

```
A>BASICA TTY/C:512
```

1.5 Files

A file is a collection of data not stored in the computer's random access memory (RAM). Instead, the data is stored on disk or some other device. Several commands save and retrieve file information, such as SAVE, LOAD, and LIST.

VBASICA supports the concept of device-independent I/O (input/output) files. Consequently, you can treat any type of input/output as I/O to a file—whether you are using a diskette or a printer, or whether you are communicating with a device.

1.5.1 File Specification

The physical file is described by its filename and specification, or filespec. The filespec can include pathnames. A simple filename is a sequence of characters that can optionally be preceded by a drive designation and followed by an extension.

The filespec format is:

[< device >][< pathname >][< filename >]

< pathname > is the sequence of directory names as described in Chapter 1.5.2.

< device > is a physical device:

KYBD:	Keyboard	Input only
SCRN:	Video Display	Output only
LPT1:	First Line Printer	"
LPT2:	Second Line Printer	"
LPT3:	Third Line Printer	"
COM1:	RS-232-C Port A	Input/Output
COM2:	RS-232-C Port B	"
A: to O:	Disk Drives	"

< filename > is the name given to the file. The name conforms to the VBASICA filename conventions and consists of two parts separated by a period:

< filename > [. < extension >]

The filename can have from 1 to 8 characters and the extension can have from 0 to 3 characters.

VBASICA uses a default extension of .BAS on LOAD, SAVE, RUN, MERGE, CHAIN, BLOAD, and BSAVE commands if no period appears in the filename and if the filename has fewer than 9 characters. If you do not specify the device, the current VBASICA default disk drive is assumed.

File specification for communications devices requires that you specify such items as baud rate and parity. See the OPEN COM statement for details.

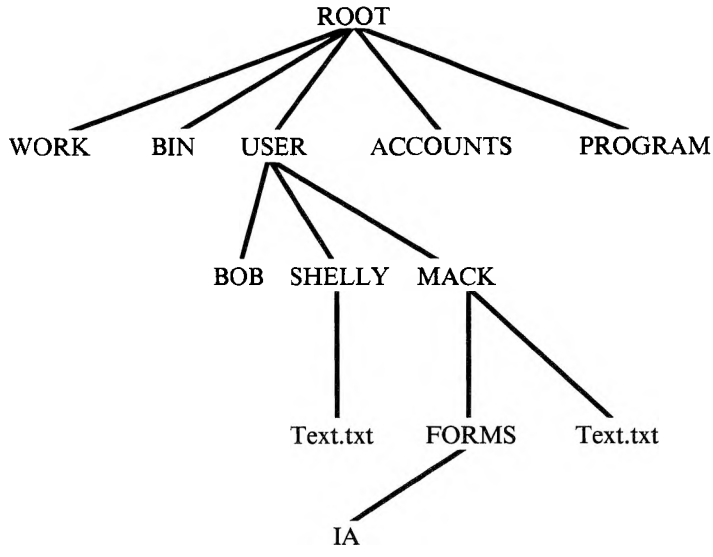
The following statements, commands, and functions support device-independent I/O. For more information, see the descriptions in Chapter 3.

BLOAD	INPUT$	LPOS	PRINT USING
BSAVE	KILL	LPRINT	PUT
CHAIN	LINE	MERGE	RESET
CLOSE	LIST	NAME	RUN
EOF	LLIST	OPEN	SAVE
FILES	LOAD	OPEN COM	WIDTH
GET	LOC	POS	WRITE
INPUT	LOF	PRINT	

1.5.2 Pathnames

A pathname is a sequence of directory names followed by a simple filename, each separated from the previous one by a backslash (\), and no longer than 63 characters. Specify the device, if any, at the beginning of the pathname. The pathname format is:

[< d > :][[\] < directory > \] [< directory > \...][< filename >]



Sample Hierarchical Directory Structure

In the structure shown above, directories are in all uppercase letters. The two entries named Text.txt and the entry named IA are files.

If a pathname begins with a backslash, MS-DOS searches for the file beginning at the root (or top) of the tree. Otherwise, MS-DOS begins at the user's current directory, known as the working directory, and searches downward from there.

\USER\SHELLY\TEXT.TXT is the pathname of Shelly's Text.txt file.

When you are in your working directory, a filename and its corresponding pathname may be used interchangeably. The following are some sample names:

\
indicates the root directory.

\PROGRAMS
Sample directory under the root directory containing program files.

\USER\MACK\FORMS\IA
A typical full pathname. This example is a file named IA in the directory named FORMS belonging to a subdirectory of USER named MACK.

USER\SHELLY
A relative pathname; it names the file or directory SHELLY in subdirectory USER of the working directory. If the working directory is the root (\), it names \USER\SHELLY.

Text.txt
Name of a file or directory in the working directory.

MS-DOS provides special shorthand notations for the working directory and the parent directory (one level up) of the working directory:

. (one period)
MS-DOS uses this shorthand notation to indicate the name of the working directory in all hierarchical directory listings. MS-DOS automatically creates this entry when you make a directory.

.. (two periods)
The shorthand name of the working directory's parent directory.

If you type the following command, MS-DOS lists the files in the parent directory of your working directory:

```
DIR ..
```

If you type the next command, MS-DOS lists the files in the parent's parent directory:

```
DIR ..\..
```

Working with Pathnames in VBASICA

Not only can VBASICA provide the ability to access files from other directories using pathnames, but you can also use it to create, change, and remove paths, using the VBASICA commands MKDIR, CHDIR, and RMDIR. For example:

- The VBASICA statement MKDIR "ACCOUNTS" creates a new directory, ACCOUNTS, in the working directory of the current drive.
- The VBASICA statement CHDIR "B:EXPENSES" changes the current directory on drive B to EXPENSES.
- The VBASICA statement RMDIR "CLIENTS" deletes an existing directory, CLIENTS, as long as that directory was empty of all files with the exception of "." and "..".

For further information on using paths in VBASICA, see the CHDIR, ENVIRON, ENVIRON$, MKDIR, and RMDIR statements in Chapter 3.

1.6 Re-Direction of Standard Input and Standard Output

VBASICA can be re-directed to read from standard input and write to standard output by providing the input and output filenames on the command line:

BASICA [program name] [< input file] [> output file]

Note that the characters "<" before the input file and ">" before the output file are literally those characters, and not angle brackets indicating a required argument. If two greater-than characters (>>) appear before the output filename, the output is appended to that file.

RULES:

1. When redirected, all INPUT, LINE INPUT, INPUT$, and INKEY$ statements will read from the input file.
2. If the program does not specify a file number in a PRINT statement, that output is redirected to the declared output file instead of the standard output device, the screen.
3. Error messages go to standard output.
4. File input from "KYBD:" still reads from the keyboard.
5. File output to "SCRN:" still outputs to the screen.
6. VBASICA continues to trap keys from the keyboard when you use the ON KEY(n) statement.
7. Pressing CTRL and PrtSc simultaneously does not cause LPT1: echoing if standard output is redirected.
8. Typing CTRL-BREAK causes VBASICA to close any open files, issue the message "Break in line < line number > " to standard output, and exit VBASICA.
9. When input is redirected, VBASICA continues to read from this source until it detects an end-of-file character. This condition may be tested with the EOF function. If the file is not terminated by a CTRL-Z or if a VBASICA input statement tries to read past end-of-file, then any open files are closed, the message "Read past end" is written to standard output, and VBASICA terminates.

EXAMPLES:

```
BASICA MYPROG > DATA.OUT
```

Data read by INPUT and LINE INPUT continues to come from the keyboard. Data output by PRINT goes into the file DATA.OUT.

```
BASICA MYPROG < DATA.IN
```

Data read by INPUT and LINE INPUT comes from DATA.IN. Data output by PRINT continues to go to the screen.

```
BASICA MYPROG < MYINPUT.DAT > MYOUTPUT.DAT
```

Data read by INPUT and LINE INPUT now comes from the file MYINPUT.DAT and data output by PRINT goes into the file MYOUTPUT.DAT.


```
BASICA MYPROG < \SALES\JOHN\TRANS >> \SALES\SALES.DAT
```

Data read by INPUT and LINE INPUT now comes from the file \SALES\JOHN\TRANS. Data output by PRINT is appended to the file \SALES\SALES.DAT.

1.7 Graphics, Screens, and Printers

VBASICA can create high-resolution graphics on the color screen. The standard monochrome screen supports text but it does not support graphics; color will support graphics, text, and color. If your computer has a color board, you can put color and black and white images on an attached color screen. VBASICA can work on either a monochrome or color screen. If you have both monochrome and color, use the MS-DOS MODE command to select either monochrome or color before you start VBASICA.

After you are in VBASICA, you can use the SCREEN statement to change the resolution mode. Three screen modes exist:

- Mode 0: A text-only video display mode.
- Mode 1: A medium-resolution mode for graphics and text.
- Mode 2: A high-resolution mode for graphics and text.

You can produce a printout (or hard copy) of a screen display in VBASICA. If you are in Graphics mode (mode 1 or 2), GRAPHICS.COM is required; you must load it before entering VBASICA. If you are in Text mode (mode 0), GRAPHICS.COM is not required. GRAPHICS.COM works only with Epson MX and FX series-compatible printers. To produce a hardcopy of your screen display, press the Shift and PrtSc (Print Screen) keys simultaneously.

The VBASICA statements that draw and manipulate images are the following:

PSET	CIRCLE	PAINT
PRESET	GET	DRAW
LINE	PUT	

You can also use the POINT function in graphics. For more information on each of these commands, see Chapter 3. The SCREEN statement describes how to choose a mode, and the COLOR statement discusses the use of colors.

1.7.1 The Color Attributes

You can specify a color attribute with the graphics statements: PSET, PRESET, LINE, CIRCLE, PAINT, and DRAW. The range is 0 to 3. These color attribute numbers are distinct from the numbers referring to actual colors; the latter are only used as parameters in the COLOR statement.

In Mode 1 on the color screen (screen 1), 0 selects the background color; 1, 2, and 3 select foreground colors.

In Mode 2 (screen 2), 0 or 2 selects black; 1 or 3 selects white.

NOTE: The COLOR statement does not affect any graphics screen except screen 1.

1.7.2 Coordinates

The drawing statements PSET, PRESET, LINE, CIRCLE, GET, PUT, and PAINT require screen locations as pairs of (x,y) coordinates. The format is (<x>,<y>), where <x> and <y> are numeric expressions. The screen coordinates are shown in Table 1-1.

1

Table 1-1: Coordinates for Color Screen

SCREEN	x (HORIZONTAL)	y (VERTICAL)
1	0-319	0-199
2	0-639	0-199

Point (0,0) is the upper left corner.

You can specify any integer coordinate value (from −32768 to 32767) for <x> and <y>. VBASICA clips out-of-range coordinates.

You can specify relative coordinates with the statements PSET, PRESET, LINE, and CIRCLE. In the following example you can write <x offset> and <y offset> as numeric expressions:

```
PSET STEP (1,1)
```

VBASICA adds their values to the current graphics cursor to determine the coordinate. The graphics cursor is the point on the screen where the last graphics point was referenced.

All the graphics statements (excluding the POINT function) update the most recent point used. If VBASICA uses the relative form on the second coordinate, it is relative to the first coordinate. In this case you can use the following:

```
STEP (<x offset>,<y offset>)
```


When you clear the screen with either the SCREEN or CLS statement, the graphics cursor is set to the middle of the screen. Table 1-2 defines the midscreen coordinates.

Table 1-2: Midscreen Coordinates for Color Screen

SCREEN	COORDINATES
Mode 1	(160,100)
Mode 2	(320,100)

1.8 Event Trapping

A program can transfer control to a specific program line when a certain event occurs with event trapping. Control is transferred as if a GOSUB statement was executed to the trap routine starting at the specified line number.

The trap routine executes a RETURN statement after completing the event. The program then resumes execution where it was before the event trap.

1.8.1 Event Specifiers

The following are event specifiers:

COM (n) n is the number of the communications channel, 1 or 2. Typically, the COM trap routine reads an entire message from the COM port before returning. **NOTE**: At high baud rates, the interrupt buffer for COM can overflow. Use the COM routine for single-character messages with discretion.

KEY (n) n is a function key number, 1–4. 1 through 10 are the soft keys 1 through 10. 11 through 14 are the cursor direction keys, as follows: 11–Up, 12–Left, 13–Right, 14–Down. A KEY (0) ON, OFF, or STOP enables, disables, or stops all 14 key events.

Note that KEY (n) ON is not the same statement as KEY ON. KEY ON displays the values of all the function keys on the twenty-fifth line of the screen.

When VBASICA traps a key, that occurrence of the key is destroyed. Therefore, you cannot use the INPUT or INKEY$ statements to find out which key caused the trap. If you want to assign different functions to particular keys, you must set up a different subroutine for each key. You cannot assign the various functions with a single subroutine.

1.8.2 Controlling Event Trapping

VBASICA controls event trapping with the following statements:

< event specifier > ON
< event specifier > OFF
< event specifier > STOP

When an event is ON and you specify a nonzero number for the trap, VBASICA checks if the specified event occurred before it starts each new statement. For example, it checks if a function key was struck or a COM character came in. If the event occurred, VBASICA performs a GOSUB to the line you specify in the ON statement.

When an event is OFF, no trapping occurs and VBASICA does not remember the event even if it occurs.

When you specify STOP, no trapping can occur. But if the event happens, VBASICA remembers it and an immediate trap occurs when VBASICA executes an < event > ON.

When you make a trap for a particular event, the trap automatically causes a stop on that event so recursive traps can never take place. The return from the trap routine automatically turns that event trap back on unless VBASICA performs an explicit OFF inside the trap routine. When an error trap occurs, it automatically disables all trapping. Trapping never takes place when VBASICA is not executing a program.

1.8.3 Additional Controls

Event trapping includes the following statements:

ON < **event specifier** > **GOSUB** < **line number** >

1

This statement sets up an event trap line number for the specified event. A < line number > of 0 disables trapping for this event.

RETURN < **line number** >

This optional form of RETURN is primarily intended for use with event trapping. The event trap routine might want to go back into the VBASICA program at a fixed line number while still eliminating the GOSUB entry that the trap created.

Use this nonlocal RETURN with care. Any other GOSUB, WHILE, or FOR active at the time of the trap remains active. If the trap returns from a subroutine, any attempt to continue loops outside the subroutine results in the "NEXT without FOR" error.

2

Data Entry and Editing in VBASICA

This chapter introduces the VBASICA character set and explains how to edit your programs with the Full Screen Editor. A description of how VBASICA handles different data types follows.

2.1 Character Set

The VBASICA character set consists of alphabetic, numeric, and special characters. The alphabetic characters in VBASICA are the upper- and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9.

2.1.1 Special Characters and Keys

Table 2-1 shows the special characters and keys VBASICA uses.

Table 2-1: Special Characters

CHARACTER	NAME
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up-arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number sign

CHARACTER	NAME
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
"	Quotation mark
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
Backspace	Deletes last character typed
Escape	Escapes edit mode subcommands
Tab	Moves print position to next tab stop (tab stops are every eight columns)
Return	Terminates a line

2.1.2 Control Characters

VBASICA uses these control characters:

- CTRL-A enters edit mode on the line being typed.
- CTRL-C interrupts program execution and returns to VBASICA command level.
- CTRL-G sounds a tone on your computer.
- CTRL-H is a backspace. It deletes the last character typed.
- CTRL-I is a tab. Tab stops are every eight columns.
- CTRL-J extends the current program line to the next physical line.
- CTRL-R retypes the current line.
- CTRL-S suspends program execution.

- CTRL-Q resumes program execution after a CTRL-S.
- CTRL-U deletes the current line.

2.2 The Full Screen Editor

Using the Full Screen Editor during program development saves you considerable time. You can learn to use the Editor by entering a sample program and practicing the edit commands described in this manual.

2.2.1 Writing Programs

The Full Screen Editor processes any line of text you type while VBASICA is in Direct mode. VBASICA is always in Direct mode after the prompt "Ok" and until you give a RUN command.

Lines of text beginning with a numeric character (digit) are considered program statements. You can extend a logical line over more than one physical line by using the linefeed key (^J). This key opens a blank line on the screen and moves any text after it down one line. A carriage return signals the end of the logical line. When you enter a carriage return the entire logical line is passed to VBASICA.

VBASICA processes program statements in one of four ways:

1. Adds a new line to the program—occurs if the line number is legal (range is 0 through 65529) and at least one nonblank character follows the line number in the line.
2. Modifies an existing line—occurs if the line number matches the line number of an existing line in the program. The text of the newly entered line replaces the existing line.
3. Deletes an existing line—occurs if the line number matches the line number of an existing line and the entered line contains ONLY a line number.

4. Produces an error.

   a. If you attempt to delete a nonexistent line, VBASICA displays the "Undefined line number" error message.

   b. If program memory is exhausted, and you add a line to the program, VBASICA displays the "Out of memory" error message and does not add the line.

You can place more than one VBASICA statement on a line, but each statement on a line must be separated from the last by a colon (:).

A VBASICA program line always begins with a line number, ends with a Return, and can contain a maximum of 255 characters.

2.2.2 Editing Programs

To edit, use the LIST command to display an entire program or range of lines on the screen, then move the cursor with the arrow keys and CTRL-B, CTRL-F, and CTRL-N to the place requiring change. Then use the Full Screen Editor for any of the following functions:

- Overtype characters.
- Delete characters to the left of the cursor.
- Delete words or characters to the right of the cursor.
- Insert characters at the cursor.
- Add, or append, characters to the end of the current logical line.

The special keys that perform these Full Screen Editor functions are described in Chapter 2.2.3.

A program line is not actually modified within the VBASICA program until you press Return. It is sometimes easier to move around the screen and make corrections to several lines; then go back to the beginning of each line you changed and press Return. The Return stores the modified lines in the program. As you make changes, the cursor might be positioned on a line containing a VBASICA message,

such as "Ok". When this happens, VBASICA automatically erases the line. The program recognizes its own messages, which are terminated by FF Hex.

It is not necessary to move the cursor to the end of the logical line before pressing Return. The Screen Line Editor remembers where each logical line ends and transfers the whole line, regardless of the cursor position.

2.2.3 Function Keys

The Full Screen Editor recognizes ten special function keys, the arrow keys, the Backspace, the Tab, and the Return. Fourteen control keys exist for moving the cursor on the screen, inserting characters, or deleting words or characters. Table 2-2 lists the keys, their corresponding hexadecimal and decimal codes, and their functions. Table 2-3 describes the functions in detail.

Table 2-2: VBASICA Function Keys

KEY			FUNCTION
01	01	CTRL-A	Edit line buffer
02	02	CTRL-B	Previous word
03	03	CTRL-C	Break (stop program)
05	05	CTRL-E	Erase to end of line
06	06	CTRL-F	Next word
08	08	CTRL-H	Destructive backspace
09	09	CTRL-I	Tab (modulo 8)
0A	10	CTRL-J	Linefeed
0B	11	CTRL-K	Home
0C	12	CTRL-L	Clear screen
0D	13	CTRL-M	Carriage return (enter logical line)
0E	14	CTRL-N	Append to end of line
12	18	CTRL-R	Toggle insert/overtype mode
14	20	CTRL-T	Display next set of function keys; toggles display on/off
15	21	CTRL-U	Clear logical line
17	23	CTRL-W	Delete word
1A	26	CTRL-Z	Clear to end of window
1C	28	→	Cursor right
1D	29	←	Cursor left
1E	30	↑	Cursor up
1F	31	↓	Cursor down
7F	128	DEL	Delete character

Table 2-3: Function Explanations

KEY	DESCRIPTION
CTRL-K HOME	Moves the cursor to the upper left corner of the screen.
CTRL-L CLEAR	Clears the screen and positions the cursor in the upper left corner of the screen.
↑	Moves the cursor up one line.
↓	Moves the cursor down one line.
←	Moves the cursor one column left. When the cursor is advanced beyond the left edge of the screen, it moves to the right side of the screen on the preceding line until it reaches the Home position.
→	Moves the cursor one position right. When the cursor is advanced beyond the right edge of the screen, it moves to the left side of the screen on the next line down until it reaches the end of the screen.
CTRL-F	Moves the cursor to the beginning of the next word. A word is defined as the characters A–Z, a–z, or 0–9, and is delineated by space characters. The next word is defined as the next character to the right of the cursor in the set [A..Z] or [0..9].
CTRL-B	Moves the cursor to the beginning of the previous word.
CTRL-N	Moves the cursor to the end of the logical line. VBASICA appends characters typed from this position to the line.
CTRL-T	Advances function key display on the 25th line to the next set of function keys, and toggles the display on/off.
CTRL-E	Erases to the end of the logical line from the current cursor position. VBASICA erases all physical lines until it finds the terminating carriage return.
CTRL-R	Toggles Insert/Overtype mode. Pressing this key changes the mode to the other mode. VBASICA automatically toggles Insert mode to Overtype mode when you press any cursor movement keys or Return. In Insert mode, VBASICA inserts typed characters at the cursor position and moves all characters on the physical line to the right. Wrap-around is in effect: characters advanced off the right edge of the screen appear from the left edge of the screen on the following line. When out of Insert mode, characters typed replace existing characters on the line.

KEY	DESCRIPTION
TAB	In Insert mode, inserts blanks from the current cursor position to the next tab stop. When out of Insert mode, moves the cursor right 8 spaces at a time until it reaches the end of the screen.
DEL	Deletes one character under the cursor for each depression. Then all characters to the right move one position left to fill in the space. If a logical line extends beyond one physical line, characters on subsequent lines are moved left one position to fill in the previous space, and move the character in the first column of each subsequent line up to the end of the preceding line.
BS	Backspace. Deletes the last character typed or the character to the left of the cursor. Moves all characters to the right of the cursor left one position. Moves subsequent characters and lines within the current logical line up, as with the DEL key.
CTRL-U	Erases the entire logical line.
CTRL-C	Returns to Direct mode, without saving any changes made to the line currently being edited.
CTRL-A	Enters the line buffer at the current cursor position for editing.
CTRL-J	Moves to the next physical line; causes scroll, if necessary.
CTRL-W	Deletes characters up to the next word.
CTRL-Z	Clears the screen to spaces from the cursor position to the end of the screen.

You can extend a logical line over more than one physical line by using the linefeed key sequence (CTRL-J). Typing a linefeed causes subsequent text to start on the next line without entering a carriage return. When a carriage return is finally entered, the entire logical line is passed to VBASICA for storage.

Occasionally, VBASICA can return to Direct mode with the cursor positioned on a line containing a message VBASICA issues, such as "Ok". When this happens the line is automatically erased.

NOTE: If you hit a carriage return at a VBASICA message, the message is sent for processing, and a syntax error results.

2.2.4 Syntax Errors

When VBASICA encounters a syntax error during program execution, VBASICA automatically enters EDIT at the line containing the error. For example,

```
10 A = 2$5              (you meant 10 A = 2^5)
RUN
Syntax Error in 10
OK
10 A = 2$5
```


The Screen Line Editor displays the line in error and puts the cursor under the digit 1. You move the cursor right to the dollar sign ($) and change it to an up-arrow (^). Then press Return. VBASICA then stores the corrected line in the program.

Variables are destroyed whenever you change a program line. If you want to examine the contents of some variable before making the change, press CTRL-C instead of moving the cursor. This action returns you to Direct mode, and preserves the variables.

2.3 Constants

Constants are the actual values VBASICA uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. The following are examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in VBASICA cannot contain commas. There are five types of numeric constants:

- Integer constants: Whole numbers between -32768 and 32767. Integer constants do not have decimal points.
- Fixed-point constants: Positive or negative real numbers; that is, numbers that contain decimal points.
- Floating-point constants: Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating-point constants is $10 < -38 >$ to $10 < 38 >$.

  Here are some examples of floating-point constants:

  ```
  235.9881E-7 = .00002359881
  2359E6 = 2359000000
  ```

  (Double-precision floating-point constants use the letter D instead of E. See Chapter 2.3.1.)

- Hex constants: Hexadecimal numbers with the prefix &H. Some examples are:

  ```
  &H76
  &H32F
  ```

- Octal constants: Octal numbers with the prefix &O or &. Examples of octal constants are:

  ```
  &0347
  &1234
  ```

2.3.1 Single- and Double-Precision Numeric Constants

Numeric constants can be single-precision or double-precision numbers. Single-precision numeric constants are stored with up to seven digits. Double-precision numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single-precision constant is any numeric constant that has seven or fewer digits, an exponential form using E, or a trailing exclamation point (!). These are examples of single-precision constants:


```
46.8
-1.09E-06
3489.0
22.5!
```

A double-precision constant is any numeric constant that has eight or more digits, an exponential form using D, or a trailing number sign (#). These are double-precision constants:

```
345692811
-1.09432D-06
3489.0#
7654321.1234
```

2.4 Variables

Variables are names that represent values in a VBASICA program. The value of a variable is assigned by the programmer, or as the result of calculations done by a program. Before a variable is assigned a value, it is assumed to have a value of zero.

2.4.1 Variable Names and Declaration Characters

VBASICA variable names can be up to 40 characters long. A variable name can contain letters, numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed.

A variable name cannot be a reserved word, but variable names can contain reserved words. Reserved words include all VBASICA commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables represent either a numeric value or a string. String variable names have a dollar sign ($) as the last character. (For example: A$ = "SALES REPORT".) The dollar sign is a variable type declaration character; it "declares" that the variable represents a string.

Numeric variable names declare integer, single-, or double-precision values. The type declaration characters for these variable names are:

%	Integer variable
!	Single-precision variable
#	Double-precision variable

The default type for a numeric variable name is single-precision.

Here are examples of VBASICA variable names:

PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single-precision value (the default type)

Variable types can also be declared by including the VBASICA statements DEFINT, DEFSTR, DEFSNG, and DEFDBL in the program. These statements are described in Chapter 3.

2.4.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) references a value in a one-dimension array; T(1,4) references a value in a two-dimension array. The maximum number of dimensions for an array is 285. The maximum number of elements per dimension is 32767.

2.4.3 Space Requirements

Table 2-4 shows the space requirements for variables and arrays.

Table 2-4: Space Requirements (in Bytes)

	INTEGER	SINGLE-PRECISION	DOUBLE-PRECISION
Variable	2	4	8
Array (per element)	2	4	8

Strings need three bytes overhead, plus the present contents of the string.

2.5 Type Conversion

When necessary, VBASICA converts a numeric constant from one type to another. You should note the following rules and examples.

If a numeric constant of one type is set equal to a numeric variable of a different type, the number is stored as the type declared in the variable name. For example,


```
10 A% = 23.42
20 PRINT A%
RUN
23
```

If a string variable is set equal to a numeric value (or vice versa), a "Type mismatch" error occurs.

During expression evaluation, all operands in an arithmetic or relational operation are converted to the same degree of precision (that is, the degree of the most precise operand). The result of an arithmetic operation is returned to this degree of precision.

In this example, the arithmetic is done in double-precision, and the result is returned in D# as a double-precision value:

```
10 D# = 6#/7
20 PRINT D#
RUN
 .857142857142857 1
```

In this example, the arithmetic is done in double-precision, and the result is returned in D (a single-precision variable), rounded, and printed as a single-precision value:

```
10 D = 6#/7
20 PRINT D
RUN
 .8571429
```

Logical operators (see Chapter 2.6.3) convert their operands to integers and return an integer result. Operands must be in the range −32768 to 32767; otherwise, an overflow error occurs.

When a floating-point value is converted to an integer, the fractional portion is rounded. For example:

```
10 LET C% = 55.88
20 PRINT C%
RUN
 56
```


If a double-precision variable is given a single-precision value, only the first seven digits (rounded) of the converted number are valid. Consequently, a single-precision value has only seven digits of accuracy. The absolute value of the difference between the printed double-precision number and the original single-precision value is less than 6.3E−8 times the original single-precision value. For example:

```
10 LET A = 2.04
20 LET B# = A
30 PRINT A; B#
RUN
 2.04 2.039999961853027
```

2.6 Expressions and Operators

An expression can be a string or numeric constant, or a variable; or it can combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. VBASICA operators are divided into these categories:

2

- Arithmetic
- Relational
- Logical
- Functional

2.6.1 Arithmetic Operators

The order of precedence of arithmetic operators is shown in Table 2-5.

Table 2-5: Arithmetic Operators

OPERATOR	OPERATION	SAMPLE EXPRESSION
^	Exponentiation	X^Y
−	Negation	−X
*, /	Multiplication, floating-point division	X*Y, X/Y
+, −	Addition, subtraction	X+Y

Use parentheses to change the order in which the operations are done. Operations within parentheses are done first. Inside parentheses, the usual order of operations is maintained. Table 2-6 shows some sample algebraic expressions and their VBASICA equivalents.

Table 2-6: Algebraic Expressions

ALGEBRAIC EXPRESSION	VBASICA EXPRESSION
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
X(Y/Z)	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X(-Y)	X*(-Y)

NOTE: Two consecutive operators must be separated by parentheses.

Integer Division and Modulus Arithmetic

Two additional operators available in VBASICA are integer division and modulus arithmetic.

Integer division is indicated by the backslash (\). The operands are rounded to integers (in the range −32768 to 32767) before the division is done. The quotient is truncated to an integer; that is, all decimal places are dropped.

Here are two examples of integer division:

```
10\4 = 2
25.68\6.99 = 3
```

In an expression, integer division is performed after all multiplication and floating-point division is finished.

Modulus arithmetic is indicated by the operator MOD. Modulus arithmetic gives an integer result equal to the remainder of an integer division.

This example:

```
10.4 MOD 4 = 2
```

is equivalent to 10/4 = 2 with a remainder of 2, but this example:

```
25.68 MOD 6.99 = 5
```

is equivalent to 26/7 = 3 with a remainder 5.

The precedence of modulus arithmetic is just after integer division.

Overflow and Division by Zero

If VBASICA encounters a division by zero while evaluating an expression, the "Division by zero" error message is displayed. Machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed. Positive machine infinity is supplied as the result of the exponentiation, and execution continues.

2.6.2 Relational Operators

Relational operators compare two values. The result of the comparison is either true (− 1) or false (0). This result can be used to make a decision regarding program flow. Table 2-7 shows the VBASICA relational operators.

Table 2-7: VBASICA Relational Operators

OPERATOR	RELATION TESTED	EXPRESSION
=	Equality	X = Y
< >	Inequality	X < > Y
<	Less than	X < Y
>	Greater than	X > Y
< =	Less than or equal to	X < = Y
> =	Greater than or equal to	X > = Y

The equal sign is also used to assign a value to a variable.

When arithmetic and relational operators are combined in one expression, the arithmetic is always done first. For example, this expression is true if the value of X plus Y is less than the value of T − 1 divided by Z:

```
X + Y < (T - 1)/Z
```

2.6.3 Logical Operators

Logical operators test multiple relations, bit manipulation, or Boolean operations. The logical operator returns a single-bit result which is either zero (false) or nonzero (true). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Table 2-8. The operators are listed in order of precedence.

Table 2-8: Logical Operations

VALUES OF X AND Y	NOT X	NOT Y	X AND Y	X OR Y	X XOR Y	X EQV Y	X IMP Y
X = 0	−1						
X = 1	0						
Y = 0		−1					
Y = 1		0					
X = 1 Y = 1	−2	−2	−1	1	0	−1	−1
X = 1 Y = 0	−2	−1	0	−1	−1	0	0
X = 0 Y = 1	−1	−2	0	−1	−1	0	−1
X = 0 Y = 0	−1	−1	0	0	0	−1	−1

Just as relational operators are used to make decisions about program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision. For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```


Logical operators convert their operands to 16-bit, signed, two's complement integers in the range -32768 to $+32767$. (If the operands are not in this range, an error results.) If both operands are 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion (that is, each bit of the result is determined by the corresponding bits in the two operands).

You can use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator can "merge" two bytes and create a particular binary value. (The two's complement of any integer is the bit complement plus one; that is, NOT X = $-$(FX = 1).) For example:

```
63 AND 16=16
```

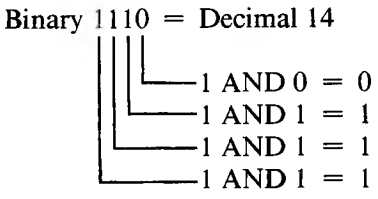
63 can be expressed as binary 111111, and 16 is binary 10000, so 63 AND 16 equals 16.

In this example:

```
15 AND 14=14
```

the answer is computed like this:

Decimal 15 = Binary 1111
Decimal 14 = Binary 1110

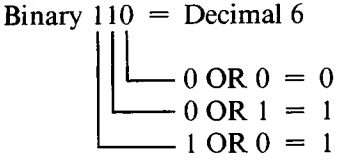


The answer to this expression:

```
4 OR 2=6
```

is computed in this way:

Decimal 4 = Binary 100
Decimal 2 = Binary 010

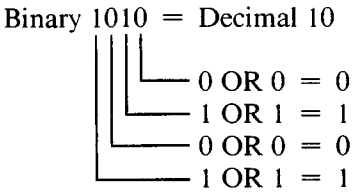


This expression:

```
10 OR 10=10
```

is computed:

Decimal 10 = Binary 1010
Decimal 10 = Binary 1010



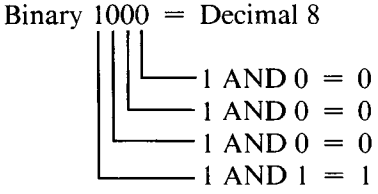
Computing logical operations with negative numbers is a little different. Each negative integer must first be converted into its bit complement, and then into its two's complement. Because each word has 16 bits, both of these complements have 16 digit places.

For example, to compute:

```
-1 AND 8=8
```

first convert -1 to its bit complement, binary 1111111111111110. Then add 1 to the bit complement to get the two's complement, 1111111111111111. Then:

Binary 1111111111111111
Decimal 8 = Binary 1000



In this example:

```
-1 OR -2=-1
```

the bit complement of −1 is 1111111111111110. Add 1 to get the two's complement, 1111111111111111.

The bit complement of −2 is 1111111111111101, and the two's complement is 1111111111111110. Then:

Binary 1111111111111111
Binary 1111111111111110
1111111111111111 = Decimal −1

2.6.4 Functional Operators

In an expression, a function calls a predetermined operation to be done on an operand. VBASICA has functions that reside in the system, such as SQR (square root) or SIN (sine). These "intrinsic" functions are described in Chapter 3. VBASICA also allows "user-defined" functions.

2.6.5 String Operations

Strings are concatenated using the plus sign (+). For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings are compared by using the same relational operators that are used with numbers:

```
=   <   >   <>   <=   >=
```

String comparisons are made by taking one character at a time from each string and comparing their ASCII codes. If all the ASCII codes are identical, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Here are some examples of string comparisons:


```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL" > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/85" where B$ = "8/12/85"
```

String comparisons can test string values or alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

2.7 Input Editing

VBASICA lets you edit your input. Use the EDIT command to change portions of a line without retyping the entire line. EDIT has this format:

EDIT <**line**>

where <line> is the number of the line where editing is to begin.

After you type the EDIT command, VBASICA enters edit mode at the specified line, displays the line number of the line to be edited (followed by a space), and then waits for you to enter an edit mode subcommand.

To delete the entire program in memory, use the NEW command described in Chapter 3. NEW is usually used to clear memory before entering a new program.

2.7.1 Syntax Errors

When VBASICA encounters a syntax error during execution of a program, it automatically enters edit mode at the line that contains the error. For example:

```
10 K = 2(4)
RUN
Syntax error in 10
10
```

When you finish editing the line and press Return (or use the E subcommand), VBASICA reinserts the line. All variable values are lost. To preserve the variable values for examination, exit edit mode with the Q subcommand. VBASICA returns to command level, and all variable values are saved.

2.7.2 CTRL-A

Type a CTRL-A to enter edit mode on the line you are currently typing. VBASICA responds with a carriage return, and the cursor moves to the first character in the line.

If you have just entered a line and want to edit it, the command:

```
EDIT.
```

enters edit mode at the current line. (The period refers to the current line.)

2.8 Error Messages

An error message is displayed if VBASICA detects an error that causes program execution to halt.

See Appendix A for a complete list of VBASICA error codes and messages.

3

VBASICA Statements, Commands, and Functions

This chapter describes the VBASICA statements, commands, and functions. It is organized as follows:

FORMAT:

Shows the correct format for the instruction.

PURPOSE:

Tells what the instruction does.

REMARKS:

Describes in detail how to use the instruction.

EXAMPLE:

Shows sample programs or program segments that demonstrate the use of the instruction.

Wherever the format for a statement or command is given, the following rules apply:

1. Items in uppercase letters must be input as shown.
2. Items in lowercase letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (that is, commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) can be repeated any number of times (up to the length of the line).

ABS Function

FORMAT:

ABS(X)

PURPOSE:

Returns the absolute value of the expression X.

EXAMPLE:

```
PRINT ABS(7*(-5))
 35
Ok
```

ASC Function

FORMAT:

ASC(X$)

PURPOSE:

Returns a numeric value that is the ASCII code of the first character of string X$. If X$ is null, VBASICA returns an "illegal function call" error. (See Appendix C for ASCII codes.)

See the CHR$ function for numeric-to-ASCII conversion.

EXAMPLE:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

ATN Function

FORMAT:

ATN(X)

PURPOSE:

Returns the arctangent of X in radians. The result is from $-pi/2$ to $pi/2$. X can be any numeric type, but ATN is always evaluated in single-precision.

EXAMPLE:

```
10 INPUT X
20 PRINT ATN(X)
RUN
?3
 1.249046
Ok
```

3

AUTO Command

FORMAT:

AUTO [< line >][,[< increment >]]

PURPOSE:

Generates a line number automatically after each carriage return.

REMARKS:

< line > is the line number of the first program line.

< increment > is the number by which AUTO increments each subsequent line number.

AUTO starts numbering at the specified line number and increments each subsequent line number by the specified increment. The default for both parameters is 10. If a comma follows < line > and < increment > is not specified, VBASICA uses the increment specified in the most recent AUTO command.

If AUTO generates a line number already in use, an asterisk appears after the number to warn you that any input replaces the existing line. Press Return immediately after the asterisk to save the line and generate the next line number.

Press CTRL-C to terminate AUTO and return to the VBASICA command level. VBASICA does not save the line in which the CTRL-C occurs.

EXAMPLE:

```
AUTO 100,50
```

generates line numbers 100, 150, 200, and so on, in increments of 50.

```
AUTO
```

generates line numbers 10, 20, 30, 40, and so on, in increments of 10, the default increment.

BEEP Statement

FORMAT:

BEEP

PURPOSE:

Sounds the speaker at 800 Hz for ¼ second.

REMARKS:

Both BEEP and PRINT CHR$(7); have the same effect.

EXAMPLE:

```
2430 IF X < 20 THEN BEEP 'if X is out of range,
     'complain.
```

BLOAD Command

FORMAT:

BLOAD < **filespec** > [, < **offset** >]

PURPOSE:

Loads a memory image into memory.

3

REMARKS:

< filespec > is a string expression representing the file specification. It can contain a path as described in Chapter 1.5, except for the extension. If you omit the device name, VBASICA assumes the current drive. The only valid extensions are the following:

(none)	(no extension)
.B	for VBASICA programs in the internal format (created with the SAVE command)
.P	for protected VBASICA programs in the internal format (created with the SAVE ,P command)
.A	for VBASICA programs in ASCII format (created with the SAVE ,A command)
.M	for memory image files (created with the BSAVE command)
.D	for data files (created by OPEN followed by output statements)

If no period appears in the filename and if the filename has less than nine characters, VBASICA assigns the default extension .BAS.

< offset > is a numeric expression from 0 to 65535. The expression is the address at which the loading starts, specified as an offset into the segment declared by the last DEF SEG statement. If you omit the < offset >, VBASICA assumes the < offset > specified in the last BSAVE.

WARNING: BLOAD does not perform address range checking. Therefore, it is possible to BLOAD anywhere in memory. Ensure that you are not overwriting the operating system, VBASICA, or your own program.

EXAMPLE:


```
10 'Load an assembly program into VBASICA DS
20 'assuming no program has been loaded.
30 DEF SEG   'set the data segment to VBASICA's
40 BLOAD "MOVE",0 'load the CALLable program
```

BSAVE Command

FORMAT:

BSAVE < **filespec** > , < **offset** > , < **length** >

PURPOSE:

Saves portions of the computer's memory on the specified device.

REMARKS:

< filespec > is a string expression representing the file specification. Refer to Chapter 1.5 for more information on file specifications.

< offset > is a numeric expression from 0 to 65535. This expression is the address at which the saving starts, specified as an offset into the segment the last DEF SEG declared.

< length > is a valid numeric expression returning an unsigned integer from 1 to 65535, which is the length of the memory image to be saved.

EXAMPLE:

```
10 'Save the color screen buffer
15 'point segment at screen buffer
20 DEF SEG=&HB800
25 'save buffer in file PICTURE
30 BSAVE "PICTURE",0,&H4000
```

As explained in the BLOAD command, the address of the 16K screen buffer for the Color/Graphics Monitor Adapter is hex B8000. The address of the 4K screen buffer for the Monochrome Display and Parallel Printer Adapter is hex B0000.

The DEF SEG statement must be used to set up the segment address to the start of the screen buffer. Use an offset of 0 and length &H4000 to specify that the entire 16K screen buffer is to be saved.

CALL Statement

FORMAT:

CALL <variable name> [(<argument list>)]

PURPOSE:

The CALL statement is the recommended way of interfacing 8086 machine language programs with VBASICA. Do not use the outmoded user call: x = USR(n).

REMARKS:

<variable name> contains the address of the starting point in memory of the subroutine being CALLed.

<argument list> contains the variables or constants, separated by commas, to be passed to the routine.

When you invoke the CALL statement, the following occurs:

1. For each parameter in the argument list, the 2-byte offset into the data segment [DS] of the parameter's location is pushed onto the stack.
2. The return address code segment [CS] and offset [IP] are pushed onto the stack.
3. Control is transferred to your routine via the segment address given in the last DEF SEG statement and offset given in <variable name>.

Your routine now has control. You can reference parameters by moving the stack pointer [SP] to the base pointer [BP] and adding a positive offset to [BP].

RULES:

The assembly language subroutine must follow these rules to work correctly:

1. It must be declared FAR.
2. Segment registers DS and ES must be restored to their entry values before returning to VBASICA.
3. The general purpose registers (AX, BX, CX, DX, SI, DI, and BP) can have any value when returning to VBASICA.
4. The assembly language routine must not change the length of any VBASICA strings.
5. The assembly language routine must perform a RET (n), where n = 2 times the number of parameters, to restore the stack pointer to its proper value.
6. You can return values to VBASICA by passing a parameter in which the result will be returned.

VBASICA Data Types

To manipulate data passed to an assembly language subroutine, you must understand how the various data types are represented in memory. When a subroutine is called, VBASICA passes the address of one of the following data representations:

1. Integer: A two-byte, two's-complement number.
2. Single precision number: A four-byte, binary, floating-point quantity. The most significant byte contains the value of the exponent minus 127. The remaining three bytes contain the mantissa. The most significant byte of the mantissa contains the sign bit, followed by the seven highest bits of the mantissa. A positive number is represented with a 0 as the sign bit, and a negative number with a 1 as the sign bit. The decimal point is left of the most significant bit of

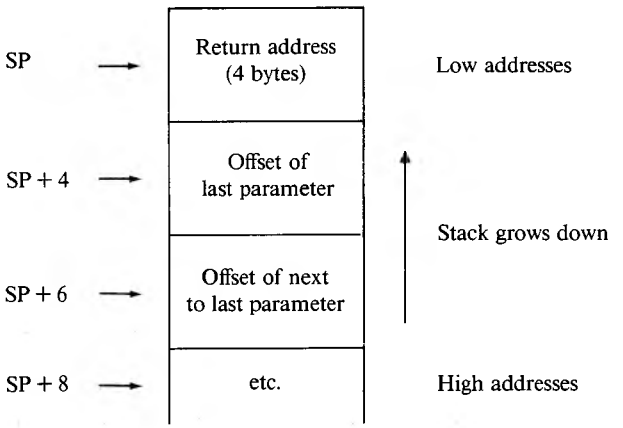
the mantissa. A 1 is always assumed to exist immediately to the right of the mantissa, although it is not represented. Thus the number is represented as the following:

```
((sign) 1.(mantissa)*2)^(exponent-127)
```

3. Double precision number: An eight-byte, binary, floating-point quantity. It is represented exactly as a single precision number, except that the mantissa consists of 55 bits, that is, 7 bytes less the sign bit.
4. String: VBASICA passes the offset address of a string descriptor, which is a three-byte data structure. The first byte of the string descriptor contains the length of the string. The next two bytes contain the address of the actual ASCII string. The assembly language subroutine can modify the string, but it must not change the string descriptor or the string length.
5. Array: Arrays consist of sequential elements of the array type. For example, an integer array containing twenty elements is represented as twenty sequential integers in memory.

Passing Parameters

VBASICA passes all subroutine parameters by reference—that is, the actual location of the parameter is passed, not a copy of its value. In a CALL statement, the offset of each parameter's address is pushed onto the stack in the same order that the parameters are listed in the procedure call. All parameters to the assembly language subroutine must be variables. Upon entry to the subroutine, the stack is arranged as shown in the following diagram. Reference the parameters by using the BP register to get their address off the stack.



EXAMPLE:

The following example shows how to load an assembly language subroutine from a VBASICA program. The assembly language routine performs modulo arithmetic on two integers, returning the remainder from dividing the first integer by the second. In this example, the assembly language module is loaded at address 1664:0 Hex, but this address is different for different applications. An explanation of the method to determine this address follows the example.

```
10 '
20 ' load the MODULO routine
30 '
40 DEF SEG = &H1664
50 BLOAD "MODULO",0
60 MODULO = 0
70 '
80 ' call the MODULO routine with some sample data
90 '
100 A% = 140
110 B% = 11
120 REMAINDER% = 0
130 CALL MODULO(A%,B%,REMAINDER%)
140 PRINT A%;"modulo";B%;"is";REMAINDER%
150 END
```

Assembly language module for use with CALL statement:

```
name        modulo

code        segment public 'code'
assume      cs:code,ds:code

modulo      proc      far

; This module is called from VBASICA with 3
; parameters, using the CALL statement.  It
; divides the first parameter by the second
; and returns the remainder in the third.

            mov     bp, sp      ;BP used to get parameters
            mov     bx, [bp+8]  ;BX=pointer to dividend
            mov     ax, [bx]    ;AX=value of dividend
            mov     bx, [bp+6]  ;BX=pointer to divisor
            mov     cx, [bx]    ;CX=value of divisor
            mov     dx, 0       ;DX:AX=dividend
            idiv    cx          ;AX=quotient,DX=remainder
            mov     bx, [bp+4]  ;BX=pointer to result
            mov     [bx], dx    ;return result to VBASICA
            ret     6           ;no. of parameters*2=6

modulo      endp
code        ends
            end
```

Loading the Assembly Language Module

To call the assembly language module, you must know its location (address). With the BLOAD statement, you can load the module at any physical address. To use the BLOAD statement to load a module, you must first create the disk file containing the module with LINK, DEBUG, and the BSAVE statement, as follows:

1. After assembling your module to create the object file, use the linker to create the .EXE file. Use the /HIGH switch when linking so the module loads in high address memory.

2. Use the debugger to load the .EXE file produced in step 1.
3. Display the register values with the R command to determine where the subroutine was loaded. Write down the values contained in the CS:IP register pair and the CX register. The CS:IP register pair contains the starting address of the subroutine and the CX register contains its length.
4. Load and execute VBASICA from DEBUG with this sequence of commands:

   NVBASICA.EXE
   L
   N
   G

   Your assembly language module is still loaded in high address memory.
5. Set the segment value in VBASICA with a DEF SEG statement:

   DEF SEG = <value in CS register>

   These values are hexadecimal and must be preceded with &H.
6. Save the module with a BSAVE statement:

   BSAVE <filespec>, <value in IP reg.>, <value in CX reg.>

You can now call the assembly language subroutine from your VBASICA program. Your VBASICA program requires the following statements before you can call the subroutine:

```
DEF SEG = <value in CS register>
BLOAD <filespec>, <value in IP register>
<SUBROUTINE> = <value in IP register>
```

You can then call the subroutine with statements of the form:

```
CALL <SUBROUTINE> <PARAMETER1, PARAMETER2>, ...
```

NOTE: You may need to use the /M: switch to set the top of VBASICA's DS at your CS-1 to keep from loading your routine on top of VBASICA.

CDBL Function

FORMAT:

CDBL(X)

PURPOSE:

Converts X to a double-precision number.

EXAMPLE:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67    454.6699829101563
Ok
```


CHAIN Statement

FORMAT:

CHAIN [MERGE] < filespec > [,[< line number expr >][,[ALL]
[,DELETE < range >]]]

PURPOSE:

Calls a program and passes variables to it from the current program.

REMARKS:

< filespec > is a string expression representing the file specification. Refer to Chapter 1.5 for more information on file specifications.

< line number expr > is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If you omit < line number expr >, execution starts at the first line.

< line number expr > is not affected by a RENUM command. With the ALL option, VBASICA passes every variable in the current program to the called program. If you omit the ALL option, the current program must contain a COMMON statement to list the passed variables.

If you use the ALL option, and not < line number expr >, a comma must hold the place of < line number expr >.

EXAMPLE:

```
CHAIN"PROG1",1000,ALL
```

If you include the MERGE option, VBASICA brings a subroutine into the program as an overlay. That is, a MERGE operation occurs with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. For example,

```
CHAIN MERGE"OVRLAY",1000
```

Delete an existing overlay each time a new overlay is brought in with the DELETE options. For example,

```
CHAIN MERGE"OVRLAY2",100, DELETE 1000-5000
```

The RENUM command affects the line numbers in < range >.

If you omit the MERGE option, CHAIN does not preserve variable types or user-defined functions for use by the chained program. You may restate any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables in the chained program.

The CHAIN statement with the MERGE option leaves the files open and preserves the current OPTION BASE setting.

The VBASICA compiler does not support the ALL, MERGE, DELETE, and < line expr > options to CHAIN. Use the statement format CHAIN < filename >. If you want to maintain compatibility with the VBASICA compiler, use COMMON to pass variables and do not use overlays. The CHAIN statement leaves the files open during CHAINing.

When you use the MERGE option, put user-defined functions ahead of any CHAIN MERGE statements in the program. Otherwise, the user-defined functions are undefined after the merge is complete.

CHDIR Command

FORMAT:

CHDIR < **pathname** >

PURPOSE:

Changes the current directory.

REMARKS:

< pathname > is a string expression specifying the name of the new directory. The string expression can be a control or variable. CHDIR works exactly like the MS-DOS command CHDIR. < pathname > must be a string of less than 63 characters.

EXAMPLE:

This line makes SALES the current directory:

```
CHDIR "SALES"
```

This line changes the current directory to USERS on drive B. It does not, however, change the default drive to B:

```
CHDIR "B:USERS"
```

The following lines change the current directory to the directory DATA one level below the root directory on the current drive:

```
PATH$="\DATA"
CHDIR PATH$
```

Also see the MKDIR and RMDIR statements. See your MS-DOS manual for a complete discussion of directories.

CHR$ Function

FORMAT:

CHR$(I)

PURPOSE:

Returns a string whose one element has ASCII code I.

REMARKS:

Refer to Appendix C for a listing of ASCII codes. CHR$ is commonly used to send a special character to the screen. For example, you can send the BEL character (CHR$(7)) as a preface to an error message, or you can send a form feed (CHR$(12)) to clear a screen and return the cursor to the home position. See the ASC$ function for ASCII-to-numeric conversion.

EXAMPLE:

```
PRINT CHR$(66)
 B
Ok
```

CINT Function

FORMAT:

CINT(X)

PURPOSE:

Converts X to an integer by rounding the fractional portion. An "Overflow" error occurs if X is not in the range −32768 to 32767.

Use CDBL and CSNG to convert numbers to the double- and single-precision data type. Use FIX and INT to return integers.

EXAMPLE:

```
PRINT CINT(45.67)
 46
Ok
```

CIRCLE Statement

FORMAT:

CIRCLE (<xcenter>,<ycenter>),<radius>
[,<attribute>[,<start>,<end>[,<aspect>]]]

PURPOSE:

Draws an ellipse with center (<xcenter>,<ycenter>) and radius <radius> for graphics only.

REMARKS:

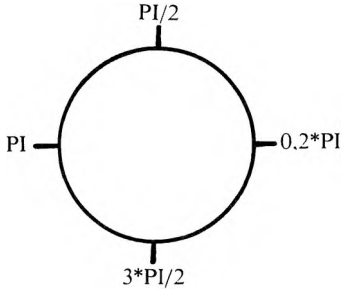
<xcenter> is an integer expression for the x-coordinate of the center of the ellipse.

< ycenter > is an integer expression for the y-coordinate of the center of the ellipse.

< radius > is an integer expression for the radius of the ellipse.

< attribute > is an expression returning the value 0 to 3; the value determines the color of the ellipse. An attribute of 0 draws an ellipse of the background color.

< start > and < end > specify where the drawing of the ellipse begins and ends. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:



If the start or end angle is negative (− 0 is not allowed), the ellipse is connected to the center point with a line. The angles are treated as if positive (not the same as adding 2*PI). The start angle can be greater or less than the end angle.

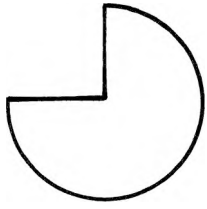
VBASICA clips points off the screen.

EXAMPLE:

The following example:

```
10 PI=3.141593
20 SCREEN 1
30 CIRCLE (160,100),60,,-PI,-PI/2
```

draws a part of a circle similar to the following:



< aspect > is the ratio of the x radius to the y radius. The default aspect ratio is 5.25/8.00 in high-resolution, and gives a visual circle, assuming a standard screen aspect ratio.

CLEAR Command

FORMAT:

CLEAR [,[< expr1 >][, < expr2 >]]

PURPOSE:

Sets all numeric variables to zero, all string variables to null, and closes all open files. CLEAR optionally sets the end of memory and the amount of stack space.

< expr1 > is a memory location that sets the highest location available for use by VBASICA.

< expr2 > sets aside stack space for VBASICA. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

REMARKS:

VBASICA allocates string space dynamically. An "Out of string space" error occurs if no free memory is left.

VBASICA supports the CLEAR statement with the restriction that < expr1 > and < expr2 > must be integer expressions. If you give a value of 0 for either expression, VBASICA uses the appropriate default. The default stack size is 256 bytes, and the default top of memory is the current top of memory.

With VBASICA, the CLEAR statement:

- Closes all files.
- Clears all COMMON and user variables.
- Resets the stack and string space.
- Releases all disk buffers.

EXAMPLES:

```
CLEAR
CLEAR ,32768
CLEAR ,,2000
CLEAR ,32768,2000
```

CLOSE Statement

FORMAT:

CLOSE [[#]<filenum>[,[#]<filenum>]...]

PURPOSE:

Concludes I/O to a disk file.

REMARKS:

<filenum> is the number under which the file was OPENed. A CLOSE without arguments closes all open files.

The association between a particular file and file number ends when you execute a CLOSE. You can open the file with the same or a different file number, and you can reuse the file number to OPEN any file. With sequential output files, a CLOSE writes the final buffer of output. The END statement and the NEW command CLOSE all disk files automatically. STOP does not close disk files.

EXAMPLE:

This statement closes the files and devices associated with file numbers 1, 2, and 3:

```
100 CLOSE 1,#2,#3
```

CLS Statement

FORMAT:

CLS

PURPOSE:

Erases the current active screen and locates the cursor at the upper left corner of the screen.

3

REMARKS:

1. In Mode 0, the monochrome screen is cleared to white, black, or underlined, depending on the current background color. The color screen is cleared to the current background color.
2. You can also clear the screen by pressing the CTRL-L key.
3. NOTE: The SCREEN and WIDTH statements force a screen clear if the resultant screen mode created differs from the current mode.

COLOR Statement

Screen Mode 0

FORMAT:

COLOR [< foreground >] [, [< background >] [, < border >]]

PURPOSE:

The COLOR statement selects the foreground, background and border colors.

REMARKS:

< foreground > is an unsigned integer from 0 to 31 that determines the character color, with values greater than 15 blinking.

< background > is an unsigned integer from 0 to 7 that determines the color over which the character is placed. < border > is an unsigned integer from 0 to 15 that determines the color around the border of the screen, if you are using a color screen. If you are using the standard screen, VBASICA ignores this parameter.

On a color screen, COLOR selects colors according to Table 3-1.

Table 3-1: Colors for Color Screen

LOW INTENSITY	COLOR	HIGH INTENSITY	COLOR
0	black	8	dark gray
1	blue	9	high-intensity blue
2	green	10	high-intensity green
3	cyan	11	high-intensity cyan
4	red	12	high-intensity red
5	magenta	13	high-intensity magenta
6	brown	14	yellow
7	white	15	intense white

If < foreground > is less than 8, low-intensity colors are displayed; otherwise, high-intensity colors are displayed.

On the standard screen, the COLOR statement selects reverse video (black on white), underlined, or highlighted characters.

Table 3-2 shows the effects you can obtain with the specified combinations of foreground and background colors.

Table 3-2: Foreground/Background Combinations

FOREGROUND	BACKGROUND	EFFECT
7	0	Normal, white on black
1	0	Underline, white on black
0	7	Reverse, black on white
15	0	Highlight, white on black
9	0	Highlight, underline, white on black
8	7	Reverse highlight
0	0	Invisible (black)

1. Any values entered outside these ranges result in the "Overflow" or "Illegal function call" error. COLOR retains previous values.
2. You can omit any parameter. Omitted parameters assume the previously selected value.
3. The COLOR statement cannot end with a comma. If it does, a "Syntax Error" will result.
4. In Alpha mode, foreground color values 0 through 7 select character color, values 8 through 15 set the intensity bit, and values 16 through 31 set the blink bit for the character.

In Screen Mode 0, executing the COLOR statement affects only the colors of subsequently written characters.

EXAMPLE:

```
100 COLOR 0,7 'reverse video
110 COLOR ,0  'invisible characters
```

Screen Mode 1

FORMAT:

COLOR [< background > , < palette >]

PURPOSE:

Selects the colors displayed on the screen.

REMARKS:

NOTE: In this mode, the color statement does not affect the standard screen.

For the color screen, the various drawing statements (PSET, LINE, and so on) allow you to specify a color attribute from 0 through 3. Color attribute 0 requests the background color, and color attributes 1–3 request foreground colors. The COLOR statement determines how these numbers are mapped to actual colors on the screen.

< background > is an unsigned integer from 0 to 15. It determines the background color and the intensity of the display according to Table 3-1.

< palette > is either 0 or 1. If < palette > is 0, or even, the foreground colors are the following:

Color Attribute	Color
1	2 (or 10) green
2	4 (or 12) red
3	6 (or 14) brown

If < palette > is 1, or odd, the foreground colors are the following:

Color Attribute	Color
1	3 (or 11) cyan
2	5 (or 13) magenta
3	7 (or 15) white

In Screen Mode 1, executing the COLOR statement immediately affects the colors on the entire screen.

In this mode, COLOR selects the background color and a three-color palette, any of which can be used with the graphics statements PSET, PRESET, CIRCLE, LINE, PAINT, and DRAW.

EXAMPLE:

```
120 COLOR 4,0      'red background,
                   green/red/yellow foreground
```

Screen Mode 2

The COLOR statement is illegal in this mode.

COM Statement

FORMAT:

COM(< n >) ON

COM(< n >) OFF

COM(< n >) STOP

PURPOSE:

Enables or disables trapping of communications activity to the indicated serial port.

REMARKS:

You must execute a COM(< n >) ON statement to allow trapping by the ON COM(< n >) statement. After COM(< n >) ON, if you specify a nonzero line number in the ON COM(< n >) statement, every time VBASICA starts a new statement it checks if any characters have been input from the serial port.

If COM(< n >) is OFF, no trapping takes place and the event is not remembered even if it does take place.

If a COM(< n >) STOP statement is executed, no trapping can take place. If any data comes in through the serial port, it is remembered and an immediate trap occurs when COM(< n >) ON is executed.

EXAMPLE:

```
10 PORT.A = 1
20 COM(PORT.A) ON  'enable com trapping on port a
    .
    .
    .
100 COM(PORT.A) STOP 'temporarily disable trapping
    .
    .
    .
300 COM(PORT.A) ON 'enable trapping immediately
    .
    .
    .
500 COM(PORT.A) OFF 'disable trapping & forget events
```


COMMON Statement

FORMAT:

COMMON <**varlist**>

PURPOSE:

Passes variables to a CHAINed program.

REMARKS:

Use the COMMON statement with the CHAIN statement. Although COMMON statements can appear anywhere in a program, put them at the beginning.

The same variable cannot appear in more than one COMMON statement. Specify array variables with a pair of parentheses at the end of the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

EXAMPLE:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
    .
    .
    .
```

CONT Command

FORMAT:

CONT

PURPOSE:

Continues program execution after you type a CTRL-C or after VBASICA executes a STOP or END statement.

REMARKS:

Execution resumes at the point where the break occurs. If the break occurs after a prompt for an INPUT statement, the program resumes execution by displaying the prompt or prompt string.

CONT is usually used with STOP during debugging. When execution stops, you can examine and change intermediate values using Direct mode statements. Resume execution with CONT or a Direct mode GOTO, which resumes execution at a specified line number. Use CONT to continue execution after an error.

CONT is invalid if the program was edited during the break.

EXAMPLE:

```
10 INPUT A, B, C
20 K=A^2*5.3:L=8^21.26
30 STOP
40 M=C*K=100: PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```


COS Function

FORMAT:

COS(X)

PURPOSE:

Returns the cosine of X in radians. The calculation of COS(X) is done in single-precision.

EXAMPLE:

```
10 X =2*COS(.4)
20 PRINT X
RUN
 1.842122
Ok
```

CSNG Function

FORMAT:

CSNG(X)

PURPOSE:

Converts X to a single-precision number.

Use CINT and CDBL to convert numbers to the integer and double-precision data types.

EXAMPLE:

```
10 A# = 975.34217#
20 PRINT A#; CSNG(A#)
RUN
 975.34217 975.341
Ok
```

CSRLIN Variable

FORMAT:

x = CSRLIN

PURPOSE:

Returns the current line or row position of the cursor in the currently active page.

3

REMARKS:

The value returned is from 1 to 25.

x = POS(0) returns the column location of the cursor.

Refer to the LOCATE statement to see how to set the cursor line.

EXAMPLE:

```
10 ROW = CSRLIN    'Record current line.
20 COL = POS(0)    'Record current column.
30 LOCATE 24,1
40 PRINT "HELLO"   'Print HELLO on last line
50 LOCATE ROW,COL 'Restore pos. to old line, column
```

CVI, CVS, and CVD Functions

FORMAT:

CVI(< 2-byte string >)

CVS(< 4-byte string >)

CVD(< 8-byte string >)

PURPOSE:

Converts string values to numeric values. Converts numeric values read in from a random disk file from strings into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single-precision number. CVD converts an 8-byte string to a double-precision number.

EXAMPLE:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$,...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

DATA Statement

FORMAT:

DATA <list>

PURPOSE:

Stores the numeric and string constants accessed by the program's READ statement(s).

3

<list> is a list of constants containing numeric constants in any format: fixed-point, floating-point, or integer. Numeric expressions are not allowed in the list. Surround string constants with double quotation marks only if they contain commas, colons, or significant leading or trailing spaces.

REMARKS:

DATA statements are nonexecutable statements and can be put anywhere in a program. A DATA statement can contain as many constants, separated by commas, as fit on a line. A program can contain any number of DATA statements.

READ statements access a program's DATA statements in order by line number. Consequently, a series of DATA statements can be regarded as one continuous list of items, regardless of the location of each DATA statement.

The variable type, numeric or string, given in a READ statement must agree with the corresponding constant in the DATA statement.

DATA statements can be reread from the beginning by using the RESTORE statement.

EXAMPLE:

The following program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
.
.
.
80 for I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```


This program READs string and numeric data from the DATA statement in line 30:

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", "COLORADO", 80211
40 PRINT C$,S$,Z
OK
RUN
CITY          STATE        ZIP
DENVER,       COLORADO     80211
Ok
```

DATE$ Variable and Statement

FORMAT:

DATE$ = <**string expr**>	Sets the current date.
<**string expr**> = **DATE$**	Gets the current date.

PURPOSE:

Sets or retrieves the current date.

REMARKS:

<string expr> is a valid string literal or variable.

VBASICA fetches and assigns the current date to the string variable if DATE$ is the expression in a LET or PRINT statement.

VBASICA stores the date if DATE$ is the target of a string assignment.

RULES:

1. If <string expr> is not a valid string, the "Type mismatch" error results. DATE$ retains previous values.
2. For <string var> = DATE$, DATE$ returns a 10-character string in the form "mm-dd-yyyy", where mm is the month (01 to 12), dd is the day (01 to 31), and yy is the year (1980 to 2099).
3. For DATE$ = <string expr>, <string expr> can take one of the following forms:

   "mm-dd-yy"
   "mm/dd/yy"
   "mm-dd-yyyy"
   "mm/dd/yyyy"

If any of the values are out of range or missing, VBASICA issues the "Illegal function call" error. DATE$ retains any previous date.

EXAMPLE:

```
DATE$ = "10-21-82"
Ok
PRINT DATE$
10-21-1982
Ok
```

DEF FN Statement

FORMAT:

DEF FN < name > [(< parlist >)] = < func def >

PURPOSE:

Defines and names a user-written function.

REMARKS:

< name > is a legal variable name. This name, preceded by FN, becomes the name of the function.

< parlist > are the variable names in the function definition, which are replaced when VBASICA calls the function. Commas separate the items in the list.

< func def > is a one-line expression that operates the function. Variable names appearing in this expression define the function. They do not affect program variables with the same name.

A variable name in a function definition might or might not appear in the parameter list. If it does appear, VBASICA supplies the value of the parameter when the function is called. Otherwise, it uses the current value of the variable.

The variables in the parameter list represent—on a one-to-one basis—the argument variables or values given in the function call.

User-defined functions can be numeric or string. If a type is specified in the function name, VBASICA forces the value of the expression to that type before it returns it to the calling statement. A "Type mismatch" error occurs when a type specified in the function name does not match the argument type.

Execute a DEF FN statement before calling the function it defines. If you call a function before it is defined, an "Undefined user function" error occurs. DEF FN is illegal in Direct mode.

EXAMPLE:

In the following example, line 410 defines the function FNAB. The function is called in line 420:

```
.
.
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
.
.
```

DEF SEG Statement

FORMAT:

DEF SEG[= < address >]

PURPOSE:

Assigns the current segment address referenced by a subsequent CALL or POKE statement or by USR or PEEK functions.

REMARKS:

< address > is a valid numeric expression returning an unsigned integer from 0 to 65535. VBASICA saves the < address > specified for use as the segment required by the PEEK, POKE, and CALL statements.

RULES:

1. If you enter any value outside this range, an "Illegal function call" error results. DEF SEG retains the previous value.
2. If you omit the address option, the segment used is set to VBASICA's data segment. This value is the initial default.
3. If you give the address option, it should be a value based on a 16-byte boundary. For PEEK, POKE, or CALL statements, VBASICA shifts the value left four bits to form the code segment address for the subsequent call instruction. VBASICA does not perform additional checking to ensure that the resultant segment + offset value is valid.

   Refer to the POKE statement for information on the special interpretation of segment &HFFFF.
4. **NOTE**: DEF and SEG **must** be separated by a space. Otherwise, VBASICA interprets the statement DEFSEG = 100 to mean: "assign the value 100 to the variable DEFSEG."

EXAMPLE:

```
10 DEF SEG=0  'Set segment to interrupt table
20 DEF SEG    'Restore segment to VBASICA's DS.
```

DEFtype Statement

FORMAT:

DEF < **type** > < **range** >

PURPOSE:

Declares variable types as integer, single-precision, double-precision, or string.

REMARKS:

< type > is INT, SNG, DBL, or STR.

< range > is a range of letters (A–Z).

A DEF < type > statement declares that variable names beginning with the specified letter(s) are of the type specified. A type-declaration character always takes precedence over a DEF < type > statement when assigning a type to a variable.

If a program does not contain type-declaration statements, VBASICA assumes that all variables without declaration characters are single-precision variables.

EXAMPLE:

In this example, all variables beginning with the letters L, M, N, O, and P are double-precision variables:

```
10 DEFDBL L-P
```

This statement declares that all variables beginning with the letter A are string variables:

```
10 DEFSTR A
```

In the following statement all variables beginning with the letters I through N and W through Z are integer variables:

```
10 DEFINT I-N, W-Z
```


DEF USR Statement

FORMAT:

DEF USR[< **digit** > **]** = < **int expr** >

PURPOSE:

Specifies the starting address of an assembly language subroutine.

REMARKS:

< digit > is any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is specified. If you omit < digit >, VBASICA assumes DEF USR0.

< int expr > is the value of the starting address of the USR routine.

Any number of DEF USR statements can appear in a program to redefine subroutine starting addresses. This allows access to as many subroutines as necessary.

EXAMPLE:

```
.
.
.
200 DEF USR=24000
210 X=USR (Y^2/2.89)
.
.
.
```


DELETE Command

FORMAT:

DELETE [<line1>][-<line2>]

PURPOSE:

Deletes program lines.

REMARKS:

<line1> and <line2> are the numbers of two different program lines.

VBASICA always returns to command level after it executes a DELETE. An "Illegal function call" error occurs if <line1> or <line2> do not exist.

EXAMPLE:

This statement deletes line 40:

```
DELETE 40
```

This statement deletes lines 40 through 100, inclusive:

```
DELETE 40-100
```

This final statement deletes all lines up to and including line 40:

```
DELETE -40
```


DIM Statement

FORMAT:

DIM < **varlist** >

PURPOSE:

Specifies the maximum values for array variable subscripts and allocates storage accordingly.

REMARKS:

< varlist > is a list of subscripted variables.

If you use an array variable name without including a DIM statement, VBASICA assumes that the maximum value of its subscript(s) is 10. If you use a subscript greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is 0, unless specified otherwise with the OPTION BASE statement.

The DIM statement gives all elements of the specified arrays an initial value of zero.

EXAMPLE:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
   .
   .
   .
```


DRAW Statement

FORMAT:

DRAW < **string exp** >

PURPOSE:

Draws a complex object as specified by < string exp > for graphics only.

REMARKS:

< string exp > is a string expression returning a valid formatted string, using the movement commands.

DRAW combines most of the capabilities of the other graphics statements into an easy-to-use object definition language called Graphics Macro Language (GML). A GML command is a single character within a string, optionally followed by one or more characters.

Movement Commands

Each of the following movement commands begin movement from the current graphics position. This position is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen when a program is run. Refer to Chapter 1.7 for more information on screens.

Table 3-3: Movement Commands

COMMAND	ACTION
U [<n>]	Move up (scale factor * n) points
D [<n>]	Move down
L [<n>]	Move left
R [<n>]	Move right
E [<n>]	Move diagonally up and right
H [<n>]	Move diagonally up and left
G [<n>]	Move diagonally down and left
F [<n>]	Move diagonally down and right
M <x,y>	Move absolute or relative. If you precede x by + or -, VBASICA adds x and y to the current position. VBASICA connects the new point with the current position by a line. Otherwise, it draws a line from the current position to the point x,y.

NOTE: The commands move one unit if you supply no argument.

The commands listed in Table 3-4 can precede any of the movement commands.

Table 3-4: Prefixes to Movement Commands

PREFIX	ACTION
B	Move but don't plot any points.
N	Move but return to original position when done.
A <n>	Set angle n. n is from 0 to 3, where 0 is zero degrees; 1 is 90; 2 is 180; and 3 is 270. VBASICA scales figures rotated 90 or 270 degrees so that they appear the same size as with 0 or 180 degrees on a screen with the standard aspect ratio of 3 to 2.
TA <n>	Turn angle n. The value of n is from −360 to +360. If n is positive (+), the angle turns counterclockwise. If n is negative (−), the angle turns clockwise. Values entered outside of the range −360 to +360 cause an "Illegal function call" error.
C <n>	Set attribute n. n is from 0 to 3 in medium-resolution, and 0 to 1 in high-resolution.
S <n>	Set scale factor. n is from 1 to 255. VBASICA multiplies the scale factor by the distances given with U, D, L, R, or relative M commands to determine the actual distance traveled.
X <string>	Execute substring (not supported by VBASICA compiler). You can execute a second substring from a string with this command, much like GOSUB in BASIC. You can have one string execute another, which executes a third, and so on.
P <paint,boundary>	Set figure color to paint and border color to boundary. The paint parameter is an integer expression. It chooses an attribute range for the current screen mode. In medium resolution, this color is one from the current palette (0–3), defined by the COLOR statement. In high resolution, two color attributes (0 = black and 1 = white) are available. The boundary parameter is the border color of the figure to be filled, in the attribute range for the current screen mode. You must specify both paint and boundary, or an error results. This command does not support paint tiling. Numeric arguments can be constants like 123 or =variable;, where variable is the name of a variable (not supported by VBASICA compiler).

VBASICA clips points off the screen.

EXAMPLES:

To draw a box:

```
10 SCREEN 2   'must be in graphics mode
20 SIDE.LEN = 50   'set the length of each side
30 DRAW "U=SIDE.LEN;R=SIDE.LEN;D=SIDE.LEN;L=SIDE.LEN;"
```

To draw a triangle:

```
10 SCREEN 2   'must be in graphics mode
20 DRAW "E15;F15;L30"
```


EDIT Command

FORMAT:

EDIT <**line number**>

EDIT .

PURPOSE:

With the Full Screen Editor, the EDIT statement displays the line specified and positions the cursor under the first digit of the line number. You can then modify the line using the keys described in Chapter 2.2.3.

REMARKS:

<line number> is the program line number of a line existing in the program. If no such line exists, VBASICA displays line exists, the "Undefined Line Number" error message.

The period (.) always gets the last line referenced by an EDIT statement, LIST command, or error message.

END Statement

FORMAT:

END

PURPOSE:

Stops program execution, closes all files, and returns to command level.

REMARKS:

END statements can occur anywhere in the program. Unlike the STOP statement, a BREAK message does not appear with END. An END statement at the end of a program is optional. VBASICA always returns to the command level after END.

EXAMPLE:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

ENVIRON Statement

FORMAT:

ENVIRON < **string** >

PURPOSE:

Modifies a parameter in VBASICA's Environment String Table.

REMARKS:

< string > is a string expression. The value of the expression must be of the form < parameter-id > = < text >, or < parameter-id > < text >. Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right is assumed to be text.

If the parameter-id did not exist in the Environment String Table, it is appended to the end of the table. If the parameter-id exists on the table when the ENVIRON statement is executed, the existing parameter-id is deleted and the new one appended to the end of the table.

The text string is the new parameter text. If the text is a null string (""), or consists only of a semicolon (;), ENVIRON removes the existing parameter-id from the Environment String Table, and compresses the remaining body of the file.

This statement can change the PATH parameter for a child process, or pass parameters to a child by inventing a new Environment parameter. Refer to the MS-DOS PATH command.

Errors include parameters that are not strings and an "Out of memory" when no more space can be allocated to the Environment String Table. The table usually has very little free space.

EXAMPLE:

The following VBASICA command creates a default PATH to the root directory on drive A:

```
PATH=A:
```

The PATH can be changed to a new value:

```
ENVIRON "PATH=A:SALES;A:ACCOUNTING"
```

A new parameter can be added to the Environment String Table:

```
ENVIRON "SESAME=PLAN"
```

The Environment String Table now contains:

```
PATH=A:SALES;A:ACCOUNTING
SESAME=PLAN
```

If you then enter:

```
ENVIRON "SESAME=;"
```

you delete SESAME, and you have a table containing:

```
PATH=A:SALES;A:ACCOUNTING
```

Refer to the ENVIRON$ function and the SHELL command.

ENVIRON$ Function

FORMAT:

ENVIRON$(< string parameter >)

ENVIRON$(< n >)

PURPOSE:

Retrieves a parameter string from VBASICA's Environment String Table.

REMARKS:

< n > is an integer.

The string result returned by the ENVIRON$ function cannot exceed 255 characters. If a parameter name is specified, and it cannot be found or it has no following text, ENVIRON$ returns a null string. When the parameter name is specified, ENVIRON$ returns all the associated text that follows < parameter > = in the Environment String Table.

If the argument is numeric, ENVIRON$ returns the nth string in the Environment String Table. The string includes all the text, including the parameter name. If the nth string does not exist, ENVIRON$ returns a null string.

EXAMPLE:

```
100 R$=ENVIRON$("PATH")
```

Returns the current path text and stores it in string variables R$.

```
100 PRINT ENVIRON$(2)
```

Prints the second environment parameter and text on the video display.

EOF Function

FORMAT:

EOF(< filenum >)

PURPOSE:

Returns −1 (true) when the end of a sequential file is reached. Use EOF to test for end-of-file while using INPUT to avoid "Input past end" errors.

EXAMPLE:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
    .
    .
    .
```

ERASE Statement

FORMAT:

ERASE <**array list**>

PURPOSE:

Eliminates arrays from a program.

REMARKS:

Arrays can be redimensioned after they are ERASEd, or the previously allocated array space in memory can be used for other purposes. A "Duplicate Definition" error occurs if you try to redimension an array without first ERASEing it.

EXAMPLE:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

ERDEV and ERDEV$ Functions

FORMAT:

ERDEV

ERDEV$

PURPOSE:

Obtains device-specific status information.

REMARKS:

ERDEV is an integer function that contains the error code returned by the last device to declare an error. ERDEV$ is a string function that contains the name of the device driver that generated the error.

You cannot set these functions. ERDEV is set by the Interrupt X'24' handler when an error within MS-DOS is detected.

ERDEV contains the INT 24 error code in the lower eight bits.

EXAMPLE:

If a user-installed device driver, MYLPT2, runs out of paper, and the driver's error number for that problem is 9, then:

```
PRINT ERDEV, ERDEV$
```

yields:

```
9  MYLPT2
```

ERR and ERL Variables

FORMAT:

ERR

ERL

REMARKS:

The ERR and ERL variables are used in error-handling subroutines. Refer to the ON ERROR GOTO statement. ERR contains the error code for the error, and ERL contains the number of the line in which the error was detected. ERR and ERL are usually used in IF...THEN statements to direct program flow in the error-trap routine.

If the statement that caused the error was a Direct mode statement, ERL contains 65535. To test if an error occurred in a Direct statement, use the following:

```
IF 65535 = ERL THEN ...
```

Otherwise, use:

```
IF ERR = error code THEN ...
IF ERL = line number THEN ...
```

If the line number does not appear to the right of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither can appear to the left of the equal sign in a LET (assignment) statement.

ERROR Statement

FORMAT:

ERROR < **int expr** >

PURPOSE:

Simulates the occurrence of a VBASICA error or allows the user to define the error codes.

REMARKS:

< int expr > is an integer expression.

The value of < int expr > must be greater than 0 and less than 255. If the value equals that of an error message VBASICA already uses, the ERROR statement simulates the occurrence of that error, and VBASICA prints the corresponding error message.

To define your own error code, use a value greater than any used by the VBASICA error codes. (Use a very high value to prevent duplication when more error codes are added to VBASICA.) Your new error code can then be conveniently handled in an error-trap routine.

If an ERROR statement specifies a code for which no error message exists, VBASICA responds with the message "Unprintable Error." Execution of an ERROR statement for which there is no error-trap routine causes an error message to appear and execution to stop.

EXAMPLE:

In the following example, you type the second line; VBASICA responds with the third line:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in 30
```


or, in Direct mode:

```
Ok
ERROR 15
String too long
Ok
```

EXP Function

FORMAT:

EXP(X)

PURPOSE:

Returns e to the power of X. X must be less than or equal to 87.3365. If EXP overflows, the "Overflow" error message is displayed. Machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXAMPLE:

```
10 X=5
20 PRINT EXP(X-1)
RUN
 54.59815
Ok
```

FIELD Statement

FORMAT:

FIELD[#] < **filenum** > , < **width** > **AS** < **str var** > ...

PURPOSE:

Allocates space for variables in a random file buffer.

REMARKS:

< filenum > is the number under which the file was OPENed.

< width > is the number of characters to be allocated to < str var > .

< str var > is a string variable.

A FIELD statement retrieves data from a random buffer after a GET, or enters data before a PUT.

EXAMPLE:

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does not put any data in the random file buffer.

Refer to LSET/RSET and GET.

The number of bytes allocated in a FIELD statement must not exceed the record length specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. The default record length is 128.

Any number of FIELD statements can be executed for the same file. All executed statements are in effect at the same time.

Do not use a FIELDed variable name in an INPUT or LET statement if a variable name is in the random file buffer. If you execute a subsequent INPUT or LET statement that uses this variable name, the variable's pointer moves to string space.

FILES Statement

FORMAT:

FILES [< filespec >]

PURPOSE:

Prints the names of files residing on the specified disk.

REMARKS:

< filespec > includes either a filename or a pathname and optional device designation.

If you omit < filespec >, VBASICA lists all the files on the currently selected drive. < filespec > is a string formula which may contain question marks (?) or asterisks (*) as wild cards. A question mark matches any single character in the filename or extension. An asterisk matches one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. You do not need to use the asterisk to request all files on a drive. For example,

```
FILES "B:"
```

If you give a filespec with no explicit path, the current directory is the default.

EXAMPLE:

This statement shows all files on the current directory:

```
FILES
```

This statement shows all files with extension .BAS:

```
FILES "*.BAS"
```

This statement shows all files on drive B:

```
FILES "B:*.*"
```


The next example shows all five-letter files whose names start with "TEST" and end with the .BAS extension:

```
FILES "TEST?.BAS"
```

If SALES is a subdirectory of the current directory, this statement displays SALES < dir >. If SALES is a file in the current directory, this statement displays SALES:

```
FILES "\SALES"
```

This statement displays MARY < dir > if MARY is a subdirectory of SALES. If MARY is a file, the statement displays its name.

```
FILES "\SALES\MARY"
```

FIX Function

FORMAT:

FIX(X)

PURPOSE:

Returns the truncated integer part of X.

REMARKS:

FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

EXAMPLES:

```
PRINT FIX(58.75)
 58
Ok
PRINT FIX(-58.75)
 -58
Ok
```

FOR...NEXT Statement

FORMAT:

```
FOR <var> = x TO y [STEP z]
    .
    .
    .
NEXT [<var>][,<var>...]
```

PURPOSE:

Allows a series of instructions to be performed in a loop a given number of times.

REMARKS:

x, y, and z are numeric expressions.

<var> is used as a counter.

The first numeric expression (x) is the initial value of the counter; the second expression (y) is the final value of the counter. VBASICA executes the program lines after the FOR statement until it encounters the NEXT statement. Then, the counter is incremented by the amount STEP specifies. VBASICA checks if the value of the counter exceeds the final value (y). If y was not exceeded, VBASICA branches back to the statement after the FOR statement and repeats the process. If y was exceeded, execution continues with the statement following the NEXT statement. The process just described is a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is less than the initial value. VBASICA decrements the counter each time the loop executes. The loop executes until the value of the counter is less than the final value.

VBASICA skips the body of the loop if the initial value of the loop, multiplied by the sign of the step, exceeds the final value times the sign of the step.

FOR...NEXT loops can be nested; that is, one FOR...NEXT loop can be put inside another loop. Each nested loop must have a unique variable name as its counter. The NEXT statement of the inside loop must appear before that of the outside loop. If nested loops have the same end point, they can share a single NEXT statement.

If the NEXT statement references only one variable, that variable can be omitted. In this case, the NEXT statement matches the most recent FOR statement. If a VBASICA statement encounters a NEXT statement before the corresponding FOR statement, VBASICA issues a "NEXT without FOR" error message and execution stops.

EXAMPLE:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 K=K+10
40 PRINT I;
50 PRINT K
60 NEXT I
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
 Ok
```

In the following example, the initial value of the loop exceeds the final value. The loop does not execute.

```
10 I=5
20 J=0
30 FOR I=I TO J
40 PRINT I
50 NEXT I
```

In the next example, the loop executes ten times:

```
10 X=5
20 FOR X=1 TO X=5
30 PRINT X;
40 NEXT X
RUN
1  2  3  4  5  6  7  8  9  10
Ok
```

VBASICA always sets the final value of the loop variable before setting the initial value.

FRE Function

FORMAT:

FRE(0)

FRE(X$)

PURPOSE:

Arguments to FRE are dummy arguments. FRE returns the number of memory bytes VBASICA is not using.

FRE ("") forces a garbage collection before it returns the number of unused bytes. Be patient—garbage collection can take up to 90 seconds. VBASICA does not usually collect garbage until all free memory is used. By using FRE("") periodically, you will have shorter delays for each garbage collection.

EXAMPLE:

```
PRINT FRE(0)
 14542
Ok
```

GET Statement for File I/O

FORMAT:

GET [#] < file number > [, < record number >]

PURPOSE:

Reads a record from a random disk file into a random access buffer.

REMARKS:

< file number > is the number under which the file was OPENed. If < record number > is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215.

The GET and PUT statements allow fixed-length input and output for VBASICA COM files. However, because of the low performance associated with telephone line communications, we recommend that you do not use GET and PUT for telephone communication.

EXAMPLE:

```
GET #1,75
```

NOTE: After VBASICA executes a GET statement, INPUT# and LINE INPUT# may be executed to read characters from the random file buffer. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

GET and PUT Statements for COM

FORMAT:

GET < **file number** > , < **nbytes** >

PUT < **file number** > , < **nbytes** >

PURPOSE:

GET and PUT allow fixed-length I/O for COM.

REMARKS:

< file number > is an integer expression returning a valid file number.

< nbytes > is an integer expression returning the number of bytes to be transferred into or out of the file buffer. nbytes cannot exceed the value set by the /S: switch when VBASICA was invoked.

Because of the low performance associated with telephone line communication, you should not use GET and PUT in such applications.

EXAMPLE:

```
10 '***  Program to Send an ASCII file over Port A
20 INPUT "Enter ASCII file to transmit:";FILNME$
30 OPEN "COM1:9600,0,7,1" AS #1     'init PORT A
40 OPEN "Output" ,#2, FILNME$, 128 'open ASCII file
50 WHILE NOT(EOF(2))
60 GET #1      'load the file buffer with a rec
70 PUT #2,128 'send the record out port A
80 WEND
90 CLOSE #1, #2
100 END
```

GET and PUT Statements for Graphics

FORMAT:

GET (< x1-coord > , < y1-coord >)-(< x2-coord > ,
< y2-coord >), < array name >

PUT (< x1-coord > , < y1-coord >),
< array > [, < action verb >]

PURPOSE:

Reads (GET) or writes (PUT) pixels to or from an area of the screen.

REMARKS:

< x1-coord > and < y1-coord > are numeric expressions returning a value in the integer range that specifies one corner of the rectangular area.

< x2-coord > and < y2-coord > are numeric expressions returning a value in the integer range that specifies the opposite corner of the rectangular area.

< array name > is a previously dimensioned array to receive the graphics points.

< array > is an array containing graphics information.

< action verb > is one of the following:

PSET, PRESET, AND, OR, XOR

The PUT and GET statements transfer graphics images to and from the screen. PUT and GET make animation and high-speed object motion possible in either Graphics mode.

The GET statement transfers the screen image into the array. The rectangle described by the specified points bounds the screen image. You define the rectangle in the same way as the rectangle drawn by the LINE statement using the ,B option.

The array is a place to hold the image and can be of any type except string. It must be dimensioned large enough to hold the entire image. The contents of the array after a GET are meaningless when interpreted directly, unless the array is of type integer.

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. The "Illegal function call" error results if the image to be transferred is too large to fit on the screen.

The action verb interacts the transferred image with the image already on the screen. PSET transfers the data onto the screen verbatim.

PRESET is the same as PSET except that VBASICA produces a negative image (black on white).

Use AND to transfer the image only if an image already exists under the transferred image. Use OR to superimpose the image onto the existing image.

XOR inverts the points on the screen where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background twice, the background is restored unchanged. Thus, you can move an object around the screen without losing the background.

NOTE: The default action mode is XOR.

It is possible to GET an image in one mode and PUT it in another, although the effect might be unusual because of the way points are represented in each mode.

AND, OR, and XOR have the following effects on color:

AND

array attr	screen attrib 0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

OR

array attr	screen attrib 0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

array attr	screen attrib 0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Animation of an object can be performed as follows:

1. PUT the object(s) on the screen.
2. Recalculate the new position of the object(s).
3. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).
4. Return to step 1, this time PUTting the object(s) at the new location.

This type of movement leaves the background unchanged. Flicker can be decreased by minimizing the time between steps 4 and 1, and by ensuring enough time delay between steps 1 and 3. If more than one object is being animated, process all objects at once, one step at a time.

If you don't have to preserve the background, you can perform animation by using the PSET action verb. Leave a border as large or larger than the maximum distance the object will move around the image when you first get it. Thus, when the object is moved, the border effectively erases any points. This method can be faster than the method using XOR because only one PUT is required to move an object (although you must PUT a larger image).

VBASICA stores the information in the array as follows:

2 bytes giving x dimension in bits
2 bytes giving y dimension
The array data itself

The data for each row of pixels is left-justified on a byte boundary. If there are less than a multiple of 8 bits stored, the rest of the byte is filled with zeros. The required array size in bytes is:

4 + INT((x* < bits/pixel > + 7)/8)*y

where < bits/pixel > is 2 in screen mode 1, and 1 in screen mode 2.

The bytes per element of an array are:

2 for integer
4 for single precision
8 for double precision

For example, if you want to GET a 10-by-12 image into an integer array, the number of bytes required is 4 + INT((10*2 + 7)/8)*12, or 40 bytes. Thus, you need an integer array with at least 20 elements.

You can examine the x and y dimensions and the data if you use an integer array. The x dimension is in element 0 of the array, and the y dimension is in element 1. Integers are stored low byte first, then high byte. The data is transferred high byte first (leftmost), then low byte.

GOSUB...RETURN Statements

FORMAT:

```
GOSUB <line>
  .
  .
  .
RETURN
```

PURPOSE:

Branches program execution to a user-defined subroutine beginning at <line>. Control returns to the main program when the subroutine finishes executing.

REMARKS:

<line> is the number of the first line of the subroutine.

In a subroutine, the RETURN statement causes VBASICA to branch back to the statement immediately after the most recent GOSUB statement. A subroutine can be called any number of times in a program. You can call a subroutine from within another subroutine. Only available memory limits the nesting of subroutines.

You can use subroutines anywhere in the program, but ensure that the subroutine is easily distinguishable from the main program. You can put a STOP, END, or GOTO statement before a subroutine to direct program control around the subroutine, and to prevent entering the subroutine by accident.

EXAMPLE:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT "IN";
60 PRINT "PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```


GOTO Statement

FORMAT:

GOTO < **line** >

PURPOSE:

Branches unconditionally out of the normal program sequence to a specified line number.

REMARKS:

< line > is the line number of the statement branched to.

If < line > is an executable statement, VBASICA executes both it and any following executable statements. If < line > is a nonexecutable statement, execution starts at the first executable statement encountered after < line > .

EXAMPLE:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
?Out of DATA in 10
Ok
```

HEX$ Function

FORMAT:

HEX$(X)

PURPOSE:

Returns a string that represents the hexadecimal value of the decimal argument. VBASICA rounds X to an integer before evaluating HEX$(X).

Use the OCT$ function for octal conversion.

EXAMPLE:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS" A$ "HEXADECIMAL"
RUN
?32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

IF Statement

FORMAT:

IF < **expr** > **THEN** < **stmnt** > | < **line** > **[ELSE** < **stmnt** > | < **line** > **]**

IF < **expr** > **GOTO** < **line** > **[ELSE** < **stmnt** > | < **line** > **]**

PURPOSE:

Makes a decision regarding program flow based on the result an expression returns.

REMARKS:

< stmt > is the statement or statements to be executed.

< line > is the line number branched to by the IF...THEN loop.

If the result of < expr > is not zero (true), the THEN or GOTO clause is executed. THEN is followed by a line number (for branching), or by one or more statements to be executed. GOTO is always followed by a line number. If the result of < expr > is zero (false), VBASICA ignores the THEN or GOTO clause and executes the ELSE clause (if present). Execution continues with the next executable statement. VBASICA allows a comma before THEN.

IF...THEN...ELSE statements can be nested. Nesting is limited only by the length of the line.

The following example is a legal statement:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

If the statement does not contain equal numbers of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. The following statement does not print "A < > C" when A < > B:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

If you are using direct mode and an IF...THEN statement is followed by a line number, an "Undefined line" error results unless you already entered a statement with the same line number while in Indirect mode.

If you use IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Perform the test against the range over which the accuracy of the value can vary.

EXAMPLE:

To test a computed variable A against the value 1.0, use the following:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E − 6.

The following statement GETs record number 1 if I is not zero:

```
200 IF I THEN GET #1,I
```

In the following example a test determines if I is greater than 10 and less than 20. If I is within this range, DB is calculated and execution resumes at line 300. If I is not within the range, execution resumes at line 110.

```
100 IF (I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In the following statement output goes to the screen or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

INKEY$ Variable

FORMAT:

INKEY$

PURPOSE:

Returns either a one-character string containing a character read from your computer, or a null string (if no character is pending). No characters are echoed. VBASICA passes all characters through to the program except for CTRL-C, which terminates the program. (With the VBASICA compiler, CTRL-C is also passed through to the program.)

EXAMPLE:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

INP Function

FORMAT:

INP(I)

PURPOSE:

Returns the byte read from port I. INP is the complementary function to the OUT statement.

REMARKS:

I is a valid machine port number from 0 to 65535.

EXAMPLE:

The following example:

```
100 A=INP(54321)
```

is equivalent to assembly language:

```
MOV         DX,54321
 IN         AL,DX
```

INPUT Statement

FORMAT:

INPUT[;][< "prompt" > {;,}] < varlist >

PURPOSE:

Allows keyboard input during program execution.

3

REMARKS:

< "prompt" > is the text of a screen prompt.

< varlist > is a list of variables that will receive input data.

When VBASICA reaches an INPUT statement, program execution pauses and VBASICA displays a question mark to indicate that you must enter data. If < "prompt" > is included in the INPUT statement, a prompt string appears before the question mark. Execution resumes after you type the required data.

To suppress the question mark, put a comma after the prompt string instead of a semicolon.

For example, the following statement prints "ENTER BIRTHDATE:" without a question mark:

```
INPUT "ENTER BIRTHDATE:",B$
```

If INPUT is followed immediately by a semicolon, then the next INPUT or PRINT statement is on the same line. VBASICA assigns the data you enter to the variables in < varlist >. The number of data items you supply must match the number of variables in the list. Separate data items by commas.

The variable list can contain numeric or string variable names, including subscripted variables. The type of each input data item must agree with the type specified by the variable name. You need not assign strings in an INPUT statement with quotation marks.

If you respond to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, for example), the message "?Redo from start" appears. Input values are not assigned until you make an acceptable response.

EXAMPLE:

In these examples, you must enter data before the program finishes executing:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?5
 5 SQUARED IS 25
Ok
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS" ;R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS" ;A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464
```

INPUT# Statement

FORMAT:

INPUT# < filenum > , < varlist >

PURPOSE:

Reads data items from a sequential disk file and assigns those items to program variables.

3

REMARKS:

< filenum > is the number used when the file is OPENed for input.

< varlist > contains the variable names assigned to the items in the file. The variable type must match the type the variable name specifies.

INPUT# does not prompt you with a question mark. The data items in the file should appear just as they would if you were typing data in response to an INPUT statement. VBASICA ignores leading spaces, carriage returns, and linefeeds when used as part of numeric values. VBASICA assumes the first character that is not a space, carriage return, or linefeed is the start of a number. The number must end on a space, carriage return, linefeed, or comma.

VBASICA also ignores leading spaces, carriage returns, and linefeeds when scanning the sequential data file for a string item. The first character that is not a space, carriage return, or linefeed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item consists of all characters read between the first quotation mark and the second. A quoted string cannot contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and terminates on a comma, carriage return, or linefeed (or after reading 255 characters). If end-of-file is reached while a numeric or string item is being input, the item is terminated.

INPUT$ Function

FORMAT:

INPUT$(X[,[#]Y])

PURPOSE:

Returns a string of X characters, from the keyboard or from file number Y. If the keyboard is used for input, no characters are echoed. All Control characters are passed through except CTRL-C, which interrupts the execution of the INPUT$ function.

EXAMPLE:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE
  IN HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
     .
     .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
     .
     .
     .
```

INSTR Function

FORMAT:

INSTR([I,]X$,Y$)

PURPOSE:

Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found.

3

REMARKS:

I is an optional offset in the range 1 to 255. X$ and Y$ are string variables, string expressions, or string literals.

Optional offset I sets the position where the search starts. INSTR returns 0 if I is less than LEN($), X$ is null, or if Y$ cannot be found. If Y$ is null, INSTR returns I or 1. If you set I equal to 0, VBASICA returns the "Illegal function call in < line > " error message.

EXAMPLE:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2   6
Ok
```

INT Function

FORMAT:

INT(X)

PURPOSE:

Returns the largest integer less than or equal to X.

See also the FIX and CINT functions. Both also return integer values.

EXAMPLES:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

IOCTL Statement

FORMAT:

IOCTL [#] < filenumber > , < string >

PURPOSE:

Transmits a control character or string to a device driver.

REMARKS:

IOCTL commands are usually two to three characters followed optionally by an alphanumeric argument. An IOCTL command string can be up to 255 bytes long.

The IOCTL statement works only if the following occur:

1. The device driver is installed.
2. The device driver states it processes IOCTL strings.
3. VBASICA performs an OPEN on a file on that device.

Most standard MS-DOS device drivers don't process IOCTL strings. You must determine whether the specific driver can handle the command.

EXAMPLE:

If you want to set the page length to 66 lines per page on LPT1, follow this procedure:

```
10 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
20 IOCTL #1, "PL66"
```

Also see the IOCTL$ function.

IOCTL$ Function

FORMAT:

IOCTL$ ([#] < filenumber >)

PURPOSE:

Receives a control data string from a device driver.

REMARKS:

The IOCTL$ function receives acknowledgment that an IOCTL statement succeeded or failed, or obtains current status information.

IOCTL$ can ask a communications device to return the current baud rate, information on the last error, and logical line width.

The IOCTL$ function works only if the following occur:

1. The device driver is installed.
2. The device driver states that it processes IOCTL strings.
3. VBASICA performs an OPEN on a file on that device.

EXAMPLE:

This example tells the device that the data is raw:

```
10 OPEN "\DEV\FOO" AS #1
20 IOCTL #1, "RAW"
```

In this continuation, if the Character Driver FOO responds "false" from the raw data mode IOCTL statement, then the file is closed:

```
30 IF IOCTL$(1) = "0" THEN CLOSE 1
```

Also see the IOCTL statement.

KEY Statement

FORMAT:

KEY <key number>,<string expression>

KEY LIST

KEY ON

KEY OFF

3

PURPOSE:

The KEY statement allows function keys to be designated as soft keys. Any one or all of the special function keys can be assigned a 15-byte string, which is input to VBASICA when the key is pressed.

Initially, the soft keys are assigned the following values:

F1 = LIST	F2 = RUN
F3 = LOAD"	F4 = SAVE"
F5 = CONT(cr)	F6 = ,"LPT1:"(cr)
F7 = TRON(cr)	F8 = TROFF(cr)
F9 = KEY	F10 = SCREEN 0,0,0(cr)

REMARKS:

<key number> is the key number, an expression returning an unsigned integer in the range 1 to 10.

<string expression> is the key assignment text, any valid string expression up to 15 characters in length.

KEY ON — The initial setting displays the key values on the 25th line. Displays only the first 6 characters of each value A carriage return in the string is indicated by <.

KEY OFF — Erases the soft key display from the 25th line.

KEY LIST	Lists all ten soft key values on the screen. Displays all 15 characters of each value.
CTRL-T	Displays next set of function keys, and toggles the display off and on.

In addition to defining soft keys (F1–F10), you can trap any shifted or unshifted key by defining it with the statement:

KEY n, CHR$ (< shift state >) + CHR$ (< scan code >)

where n is from 15 to 20.

< shift state > is a value that corresponds to the hex value for the current shift keys. Shift state values must be in hexadecimal.

Caps Lock	&H40
Num Lock	&H20
ALT	&H08
CTRL	&H04
Shift	&H01, &H02, &H03

You can use combinations of shift states; for example, CHR$(&H0C) represents CTRL-ALT together.

< scan code > is a value from 1 to 83 that represents the key to be trapped. It is the number of the key on the keyboard, not the ASCII value generated by pressing the key.

RULES:

1. If the value returned for < key number > is not from 1 to 10, VBASICA displays the "Illegal function call" error. VBASICA retains the previous key string assignment.
2. The key assignment string can be 1 to 15 characters in length. If the string is longer, VBASICA assigns the first 15 characters.
3. Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

4. When a soft key is assigned, the INKEY$ function returns one character of the soft key string per invocation. If VBASICA disables a soft key, it returns the code given for that key (see Appendix B).

5. The four cursor movement keys (up, left, down, and right) are predefined as function keys 11, 12, 13, and 14, respectively. Therefore, trapping scan codes 72, 75, 77 and 80 serve no useful purpose (they are already trappables).

EXAMPLE:

This statement displays the soft keys on the 25th line:


```
50 KEY ON
```

This statement erases soft key display:

```
200 KEY OFF
```

The following statement assigns the string 'MENU'(cr) to soft key 1. Such assignments can be used for rapid data entry. This example might be used in a program to select a menu display.

```
10 KEY 1,"MENU"+CHR$(13)
```

This statement erases soft key 1:

```
20 KEY 1," "
```

The following routine initializes the first five soft keys:

```
10 KEY OFF     'Turn off key display during init
20 DATA KEY1, KEY2, KEY3, KEY4, KEY5
30 FOR I=1 TO 5
40 READ SOFTKEYS$(I)
50 KEY I,SOFTKEYS$(I)
60 NEXT I
70 KEY ON      'now display new softkeys.
```

KEY(n) Statement

FORMAT:

KEY(< n >) ON

KEY(< n >) OFF

KEY(< n >) STOP

PURPOSE:

Activates and deactivates trapping of the specified key.

REMARKS:

< n > is a numeric expression returning a value between 1 and 20 and indicates the key to be trapped.

1–10	Function keys 1 to 10
11	Up arrow
12	Left arrow
13	Right arrow
14	Down arrow
15–20	Keys defined by the form:

KEY(n),CHR$(KBflag) + CHR(scan code)

Keys 15–20 can be trapped.

VBASICA must execute a KEY(< n >) ON statement to activate trapping of function key or cursor control key activity. After KEY(< n >) ON, if you specify a nonzero line number in the ON KEY(< n >) statement, every time VBASICA starts a new statement it checks if the specified key was pressed. If so, it performs a GOSUB to the line number specified in the ON KEY(< n >) statement.

If KEY(< n >) is OFF, no trapping takes place and the event is not remembered even if it does take place.

If a KEY(< n >) STOP statement is executed, no trapping takes place. But if the specified key is pressed, this event is remembered, and an immediate trap takes place when KEY(< n >) ON is executed.

KEY(< n >) ON has no effect on the display of the soft key values on the 25th line.

< n > cannot be an expression.

KILL Command

FORMAT:

KILL < **filespec** >

PURPOSE:

Deletes a file from disk.

REMARKS:

< filespec > is a string expression for the file specification.

A "File already open" error occurs if you try to KILL a currently OPENed file.

You can use KILL with all disk file types: program files, random data files, and sequential data files.

EXAMPLE:

```
200 KILL "DATA1"
```

LEFT$ Function

FORMAT:

LEFT$(X$,I)

PURPOSE:

Returns a string consisting of the leftmost I characters of X$. I must be from 0 to 255. If I is greater than LEN(X$), VBASICA returns all of X$. If I is zero, VBASICA returns the null string (length zero).

EXAMPLE:

```
10 A$ = "BASIC PROGRAM"
20 B$ = LEFT$(A$,5)
30 PRINT B$
 BASIC
Ok
```

LEN Function

FORMAT:

LEN(X$)

PURPOSE:

Returns the number of characters in X$. Counts nonprinting characters and blanks.

3

EXAMPLE:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

LET Statement

FORMAT:

[LET] <**var**> = <**expr**>

PURPOSE:

Assigns the value of an expression to a variable.

REMARKS:

The word LET is optional. The equal sign suffices when assigning an expression to a variable name.

The following program fragment contains several LET statements:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
      .
      .
      .
```

The same statements can also be written as the following:

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
      .
      .
      .
```


LINE Statement

FORMAT:

LINE [(<x>,<y>)] -(<x1>,<y1>) [,[<color>][,B[F]][,style]]

PURPOSE:

Draws or removes straight lines, rectangles, and filled rectangles for graphics only.

REMARKS:

<x>, <y>, <x1>, and <y1> are valid coordinates, as described in Chapter 1.7.

< color > is an expression returning a value 0 to 3, determining the color of a line. A color of 0 draws a line in the background color. See Chapter 1.7 for more information.

< style > is a 16-bit integer mask to put points on the screen. The style option is for normal lines and boxes, but cannot be used with filled boxes (BF). Using style with BF results in a syntax error. This technique is called line styling.

LINE (x,y)-(x1,y1) draws a line from point (x,y) to point (x1,y1).

LINE -(x1,y1) draws a line from the previous graphics cursor position to the point (x1,y1).

LINE (x1,y1)-(x2,y2),B draws a rectangle with (x1,y1) as one corner and (x2,y2) as the opposite diagonal corner.

Using the B argument replaces the following four LINE commands:

```
LINE (x1,y1)-(x2,y1)
LINE (x1,y1)-(x1,y2)
LINE (x2,y1)-(x2,y2)
LINE (x1,y2)-(x2,y2)
```

,BF draws the same rectangle as ,B but also fills in the interior points with the selected attribute.

LINE (x,y)-(x1,y1),BF draws a rectangle and fills the entire rectangle. Out-of-range coordinates are clipped.

EXAMPLES:

Draw lines continuously using random attribute:

```
10 CLS
20 LINE -(RND*799,RND*399),INT(RND*4)
30 GO TO 20
```

Draw alternating pattern—line on, line off:

```
10 FOR X = 0 TO 799
20 LINE (X,0)-(X,399),X AND 1
30 NEXT
```

Draw lines continuously, using random attribute, and filling the rectangles:

```
10 CLS
20 LINE -(RND*799,RND*399),RND*2,bf
30 GO TO 20
```


LINE INPUT Statement

FORMAT:

LINE INPUT[;][" < prompt > ";] < str var >

PURPOSE:

Inputs an entire line (up to 254 characters) to a string variable without using delimiters.

REMARKS:

The prompt is a string literal that appears on your screen before input is accepted. A question mark appears only if it is part of the prompt string. VBASICA assigns everything you type from the end of the prompt until you press the Return key to < str var >. If VBASICA encounters a linefeed/carriage return sequence (in this order only) it echoes both characters. However, VBASICA ignores the carriage return, puts the linefeed into < str var >, and data input continues.

If LINE INPUT is followed by a semicolon, then the next print or input statement is put on the same line, even if you press Return.

A LINE INPUT is bypassed if you type CTRL-C. VBASICA returns to command level and the "Ok" prompt appears. Use the CONT statement to resume execution at the LINE INPUT.

LINE INPUT# Statement

FORMAT:

LINE INPUT# < filenum >, < str var >

PURPOSE:

Reads all characters in a sequential file until it reaches a carriage return. The command then skips over the carriage return/linefeed sequence and stops. The next LINE INPUT# reads all characters up to the next carriage return. Any linefeed/carriage return sequences encountered are preserved.

REMARKS:

< filenum > is the number under which the file is opened.

< str var > is the variable name to which the line is to be assigned.

LINE INPUT# is useful when each line of a data file is broken into fields, or if a VBASICA program saved in ASCII mode is being read as data by another program.

EXAMPLE:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION?" ;C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES   234,4   MEMPHIS
LINDA JONES   234,4   MEMPHIS
Ok
```


LIST Command

FORMAT:

LIST [[< line no. > [-[< line no. >]]] [, < filespec >]]

PURPOSE:

Lists a program on the screen or to other devices.

REMARKS:

< line no. > is a valid line number from 0 to 65529.

< filespec > is a valid string expression returning a valid file specification.

RULES:

1. If the optional parameter <filespec> is omitted, the specified lines are listed to the screen.
2. Listings directed to the screen by omitting <filespec> can be stopped at any time by pressing CTRL-C.
3. If the line range is omitted, the entire program is listed.
4. When the dash (-) is used in a line range, you have three options:
   a. If only the first number is given, that line and all higher numbered lines are listed.
   b. If only the second number is given, all lines from the beginning of the program through the given line are listed.
   c. If both numbers are specified, the inclusive range is listed.

EXAMPLE:

```
LIST ,"LPT1:"
```

Lists program to the Line Printer.

```
LIST 10-20
```

Lists lines 10 through 20 to the screen.

```
LIST 10- ,"SCRN:"
```

Lists lines 10 through last to the screen.

```
LIST -200
```

Lists first through line 200 to the screen.

```
LIST 1000-1045,"COM1:4800,0,5,2"
```

Lists lines 1000 through 1045 to serial port A, setting the baud rate, parity, data bits, and stop bits.

LLIST Command

FORMAT:

LLIST [{ < line number > [-[< line number >]] ⊢ < line number > }]

PURPOSE:

Lists all or part of the program currently in memory on the line printer.

REMARKS:

LLIST assumes a 132-character-wide printer.

VBASICA always returns to command level after an LLIST is executed. The options for LLIST are the same as for the LIST command.

EXAMPLE:

See the examples for the LIST command. With the exception of the last one, which addresses a device, LLIST works in a similar way.

LOAD Command

FORMAT:

LOAD <filespec> [,R]

PURPOSE:

Loads a program from the specified device into memory, and optionally runs it.

3

REMARKS:

<filespec> is a valid string expression for the file specification. Refer to Chapter 1.5 for more information on file specifications.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if you use the "R" option, the program is RUN after it is LOADed, and all open data files are kept open. Thus, you can use LOAD with the "R" option to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

EXAMPLE:

This statement allows you to enter programs from the communications port:

```
LOAD "COM1:4800,O,7,1",R
```

LOC Function

FORMAT:

LOC(< file number >)

PURPOSE:

With random disk files, LOC returns the actual record number within the file.

With sequential files, LOC returns the current byte position in the file, divided by 128.

REMARKS:

< file number > is the number under which the file was opened.

When a file is opened for APPEND or OUTPUT, LOC returns the size of the file in (bytes/128).

For a communications file, LOC(X) determines if any characters are in the input queue waiting to be read. If more than 255 characters are in the queue, LOC(X) returns 255. Because strings are limited to 255 characters, this practical limit alleviates the need to test for string size before reading data into it.

If fewer than 255 characters remain in the queue, the value returned by LOC(X) depends on whether the device was opened in ASCII or binary mode. In either mode, LOC returns the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. An attempt to read the end-of-file results in an "Input past end" error.

EXAMPLE:

```
200 IF LOC(1)>50 THEN STOP
```

LOCATE Statement

FORMAT:

3

LOCATE [< row >][,[< col >] [,[< cursor >] [,[< start >]
[, < stop >]]]]

PURPOSE:

Moves the cursor to the specified position on the active screen. Optional parameters turn the cursor on and off and define the start and stop scan lines for the cursor.

REMARKS:

< row > is the screen line number, a numeric expression returning an unsigned integer in the range 1 to 25.

< col > is the screen column number, a numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending on the screen width.

< cursor > is a Boolean value indicating whether the cursor is visible, with 0 for off, nonzero for on.

< start > is the cursor starting scan line, a numeric expression returning an unsigned integer from 0 to 31.

< stop > is the cursor stop scan line, a numeric expression returning an unsigned integer from 0 to 31.

The LOCATE statement moves the cursor to the specified position. Subsequent PRINT statements begin placing characters at this location. Optionally it can be used to turn the cursor on or off or to change the size of the cursor.

Any values entered outside these ranges result in the "Illegal function call" error. VBASICA retains previous values. You can omit any of the parameters. Omitted parameters assume the previous value.

If the start scan line parameter is given and the stop scan line parameter is omitted, stop assumes the start value. If both are omitted, the start and stop scan lines retain their previous values.

EXAMPLE:

Move to the home position in the upper left corner:

```
10 LOCATE 1,1
```

Make the cursor visible; the position remains unchanged:

```
20 LOCATE ,,1
```

The cursor position and visibility remain unchanged. Set the cursor to display at the bottom of the character starting and ending on scan line 15:

```
30 LOCATE ,,,15
```

Move to line 5, column 1, turn cursor on; cursor covers entire character cell starting at scan line 0 and ending on scan line 9:

```
40 LOCATE 5,1,1,0,9
```

NOTE: Usually, VBASICA does not print to line 25. To put things on line 25, turn off the soft key display using KEY OFF, then use LOCATE 25,1 : PRINT...

LOF Function

FORMAT:

LOF(< file number >)

PURPOSE:

Returns the number of bytes allocated to the file.

REMARKS:

< file number > is associated with a currently open file. For diskette files, LOF returns a multiple of 128. For example, if the actual file length is 257 bytes, the number 384 is returned.

For communications, LOF returns the amount of free space in the input buffer. That is, size-LOC(filnum), where size is the size of the communications buffer, defaults to 256 but can be changed with the /C: option at VBASICA initialization time.

EXAMPLE:

```
10 OPEN "DATA.FIL" AS #1
20 GET #1, LOF(1)/128
```

These statements get the last record of the file, assuming the record length is 128 bytes.

```
10 OPEN "FILE.BIG" AS #1
20 GET #1, LOF(1)/128
```

These statements get the last record of the file FILE.BIG, assuming that the file was created with a default record length of 128 bytes.

LOG Function

FORMAT:

LOG(X)

PURPOSE:

Returns the natural logarithm of X. X must be greater than zero.

EXAMPLE:

```
PRINT LOG(45/7)
 1.860752
Ok
```


LPOS Function

FORMAT:

LPOS(X)

PURPOSE:

Returns the current position of the printhead within the line printer buffer. LPOS(X) does not necessarily give the physical position of the printhead.

REMARKS:

X is a dummy argument.

EXAMPLE:

```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

FORMAT:

LPRINT [<expr list>]

LPRINT USING <string expr>;<expr list>

PURPOSE:

Prints data at the line printer.

REMARKS:

Same as PRINT and PRINT USING, except output goes to the line printer.

LPRINT assumes a 132-character-wide printer.

LSET and RSET Statements

FORMAT:

LSET <**str var**> = <**str expr**>

RSET <**str var**> = <**str expr**>

PURPOSE:

Moves data from memory to a random file buffer, in preparation for a PUT statement.

REMARKS:

If <str expr> needs fewer bytes than were allocated to the field containing <str var>, LSET left-justifies the string in the field, and RSET right-justifies the string. Spaces are used to pad the extra positions. If the string is too long for the field, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, and MKD$ functions.

EXAMPLE:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

LSET or RSET can also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This is valuable when you are formatting printed output.

MERGE Command

FORMAT:

MERGE <**filespec**>

PURPOSE:

Merges the lines from an ASCII program file into the program currently in memory.

REMARKS:

<filespec> is a string expression that returns a valid file specification. Refer to Chapter 1.5 for more information on file specifications.

EXAMPLE:

```
MERGE "COM2:4800,0,7,1"
```

MID$ Function and Statement

FORMAT:

MID$(< str expr1 > ,n[,m]) = < str expr2 >

PURPOSE:

Replaces a portion of one string with another string.

REMARKS:

n and m are integer expressions.

< str expr1 > and < str expr2 > are string expressions.

Beginning at position n, the characters in < str expr1 > are replaced by the characters in < str expr2 >. The m option can be used to specify the number of characters from < str expr2 > in the replacement. If m is omitted, all of < str expr2 > is used. However, the number of characters replaced can never exceed the original length of < str expr1 >.

EXAMPLE:

```
10 A$="DIDDY WAH WAH"
20 MID$(A$,11)="DIDDY"
30 PRINT A$
RUN
DIDDY WAH DIDDY
```

MID$ can also return a substring of a given string.

MKDIR Command

FORMAT:

MKDIR <**pathname**>

PURPOSE:

Creates a new directory.

REMARKS:

<pathname> is a string expression specifying the name of the directory to be created. MKDIR works exactly like the MS-DOS command MKDIR. The <pathname> must be a string of less than 63 characters.

EXAMPLES:

Assume the current directory is the root. This example creates a subdirectory named SALES in the current directory of the current drive:

```
MKDIR "SALES"
```

The following example creates a subdirectory named USERS in the current directory of drive B:

```
MKDIR "B:USERS"
```

Also see the CHDIR and RMDIR statements.

MKI$, MKS$, and MKD$ Functions and Statements

FORMAT:

MKI$(< int expr >)

MKS$(< single-precision expr >)

MKD$(< double-precision expr >)

PURPOSE:

Converts numeric values to string values. Any numeric value put into a random file buffer using an LSET or RSET statement must be converted to a string. MKI$ converts:

- An integer to a 2-byte string
- A single-precision number to a 4-byte string
- A double-precision number to an 8-byte string

EXAMPLE:

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
    .
    .
```

NAME Command

FORMAT:

NAME <old filename> **AS** <new filename>

PURPOSE:

Changes the name of a disk file.

3

REMARKS:

<old filename> must exist and <new filename> must not exist; otherwise an error results. The renamed file occupies the same area of disk space as it did under the old name.

EXAMPLE:

In the following example, the file formerly named ACCTS is renamed LEDGER:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

NEW Command

FORMAT:

NEW

PURPOSE:

Deletes the program in memory and clears all variables.

REMARKS:

Enter NEW at command level to clear memory before entering a new program. VBASICA always returns to command level after executing a NEW command.

OCT$ Function

FORMAT:

OCT$(X)

PURPOSE:

Returns a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Use the HEX$ function for hexadecimal conversion.

EXAMPLE:

```
PRINT OCT$(24)
 30
Ok
```

ON COM Statement

FORMAT:

ON COM(< n >) GOSUB < line number >

PURPOSE:

Sets up a line number for VBASICA to trap when information is coming into the communications buffer.

3

REMARKS:

< n > is the number of the communications port, where 1 is port A and 2 is port B.

< line number > is the starting line number of the routine to handle the information coming from the port. A line number of 0000 (zero) disables trapping of communication for the specified port.

You must execute a COM(< n >) ON statement to activate this statement for port < n >. After COM(< n >) ON, if you specify a nonzero line number in the ON COM(< n >) statement, every time VBASICA starts a new statement it checks to see if any characters have come through the specified port. If so, it performs a GOSUB to the specified line number.

If you execute COM(< n >) OFF, no trapping occurs for the port and the event is not remembered even if it does take place.

If you execute a COM(< n >) STOP statement, no trapping can occur for the port. But if VBASICA receives a character, it is remembered and an immediate trap occurs when you execute COM(< n >) ON.

When the trap occurs, VBASICA creates an automatic COM(< n >) STOP. Consequently, recursive traps can never occur. The RETURN from the trap routine automatically does a COM(< n >) ON unless an explicit COM(< n >) OFF was performed inside the trap routine.

Event trapping does not occur when VBASICA is not executing a program. When an error trap (resulting from an ON ERROR statement) occurs, all trapping is automatically disabled (including ERROR, COM, and KEY).

Before returning to the main program, the communications trap routine typically reads an entire message from the communications port. At high baud rates, the overhead of trapping and reading for each character can overflow the communications buffer. Therefore, avoid using the communications trap for single-character messages.

See the RETURN statement for more information.

EXAMPLE:

```
100 PORT. A = 1
110 PORT. B = 2
120 ON COM(PORT. A) GOSUB 500
        .
        .
        .
500 '*****     ROUTINE TO HANDLE PORT A CHRS
        .
        .
        .
550 RETURN
```

ON ERROR GOTO Statement

FORMAT:

ON ERROR GOTO < **line** >

PURPOSE:

Enables error trapping and specifies the first line of the error-handling subroutine.

3

REMARKS:

< line > is the number of the first line of an error-handling subroutine.

After error trapping is enabled, all errors detected—including direct mode errors (for example, syntax errors)—cause VBASICA to jump to the specified error-handling subroutine. If < line > does not exist, an "Undefined line" error results.

To disable error trapping, execute an ON ERROR GOTO 0 statement. Subsequent errors print an error message and halt execution. An ON ERROR GOTO 0 in an error-trapping subroutine tells VBASICA to stop and print the error message for the error that caused the trap. All error-trapping subroutines should execute an ON ERROR GOTO 0 if they encounter an error for which no recovery action exists.

If an error occurs while an error-handling subroutine is executing, the VBASICA error message is printed and execution stops. Error trapping does not occur within the error-handling subroutine.

EXAMPLE:

```
ERROR
10 ON ERROR GOTO 1000
```

ON...GOSUB and ON...GOTO Statements

FORMAT:

ON < **expr** > **GOTO** < **list** >

ON < **expr** > **GOSUB** < **list** >

PURPOSE:

Branches to one of several specified line numbers, depending on the value returned when < expr > is evaluated.

REMARKS:

< expr > is a numeric expression.

< list > is a list of line numbers.

The value of < expr > determines which line number in the list is used for branching. If the value is three, the third line number in the list is the destination of the branch. If the value is noninteger, the fractional portion is rounded.

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine. If the value of < expr > is zero, or greater than the number of items in the list (but less than or equal to 255), VBASICA continues with the next executable statement. If the value of < expr > is negative or greater than 255, an "Illegal function call" error occurs.

EXAMPLE:

```
100 ON L-1 GOTO 150,300,320,390
```

ON KEY(n) Statement

FORMAT:

ON KEY(< n >) GOSUB < line number >

PURPOSE:

Sets up a line number for VBASICA to trap when you press the specified function key or cursor control key.

3

REMARKS:

< n > is a numeric expression returning a value between 1 and 20 and indicates the key to be trapped:

1–10	Function keys 1 to 10
11	Up arrow
12	Left arrow
13	Right arrow
14	Down arrow
15–20	Keys defined by the form:

KEY(n),CHR$(KBflag) + CHR$(scan code)

Keys 15–20 can be trapped. See the KEY(n) statement for more information.

< line number > is a valid number from 0 to 65529. If < line number > is 0, VBASICA disables trapping on the specified key.

Execute a KEY(< n >) ON statement to activate trapping of function key or cursor control key activity. After KEY(< n >) ON, if you specify a nonzero line number in the ON KEY(< n >) statement, every time VBASICA starts a new statement it checks if the specified key was pressed. If the key was pressed, VBASICA performs a GOSUB to the line number specified in the ON KEY(< n >) statement.

If you execute a KEY(< n >) OFF statement, no trapping occurs for the specified key and the event is not remembered even if it does occur.

If you execute a KEY(< n >) STOP statement, no trapping occurs. However, if the specified key is pressed, VBASICA remembers this event. Consequently, an immediate trap occurs when you execute a KEY(< n >) ON.

When the trap occurs, VBASICA executes an automatic KEY(< n >) STOP. Thus, recursive traps can never take place. The RETURN from the trap routine automatically performs a KEY(< n >) ON unless an explicit KEY(< n >) OFF was performed inside the trap routine.

Event trapping does not occur when VBASICA is not executing a program. When an error trap (resulting from an ON ERROR statement) occurs, VBASICA automatically disables all trapping (including ERROR, COM, and KEY).

Key trapping may not work when you press other keys before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$. Therefore, the trap routine for each key must be different if you desire a different function.

KEY(< n >) ON has no effect on whether the soft key values are displayed on the 25th line.

See the RETURN statement for more information.

EXAMPLE:

```
100 KEY.5 = 5
110 ON KEY(KEY.5) GOSUB 500
      .
      .
      .
500 '*****     ROUTINE TO HANDLE KEY(5)
      .
      .
      .
550 RETURN
```

ON PLAY Statement

FORMAT:

ON PLAY(n) GOSUB < **linenumber** >

PURPOSE:

Branches to a specified subroutine when the music queue contains fewer than (n) notes. This statement permits continuous music during program execution.

REMARKS:

(n) is an integer expression from 1 through 32. Values outside this range result in an "Illegal function call" error.

< linenumber > is the statement line number of the PLAY event trap subroutine.

PLAY ON causes an event trap when the background music queue goes from (n) notes to (n − 1) notes.

PLAY ON enables PLAY event trapping.

PLAY OFF disables PLAY event trapping.

PLAY STOP suspends PLAY event trapping.

If you execute a PLAY OFF statement, VBASICA does not perform or remember the GOSUB.

If you execute a PLAY STOP statement, VBASICA does not perform the GOSUB until it executes a PLAY ON statement.

When an event trap occurs (that is, the GOSUB is performed), VBASICA executes an automatic PLAY STOP so that recursive traps cannot occur. The RETURN from the trapping subroutine automatically performs a PLAY ON statement unless an explicit PLAY OFF occurred inside the subroutine.

You can use the RETURN < line number > form of the RETURN statement to return to a specific line number from the trapping subroutine. Use this type of return with care because any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active, and errors such as "FOR without NEXT" might result.

RULES:

1. A PLAY event trap is issued only when playing background music (for example, PLAY "MB..). PLAY event traps are not issued when running in Music Foreground (for example, default case, or PLAY "MF..).
2. A PLAY event trap is not issued if the background music queue has already gone from having (n) to (n − 1) notes when a PLAY ON is executed.
3. If (n) is a large number, event traps occur frequently enough to diminish program execution speed.

Also see the PLAY ON, PLAY OFF, and PLAY STOP statements.

EXAMPLE:

In the following example, control branches to a subroutine when the background music buffer decreases to 7 notes.

```
100 PLAY ON
      .
      .
      .
540 PLAY "MB L1 XZITHER$"
550 ON PLAY(8) GOSUB 6000
      .
      .
      .
6000 REM, **BACKGROUND MUSIC**
6010 LET COUNT% = COUNT% + 1
      .
      .
6999 RETURN
```


ON TIMER Statement

FORMAT:

ON TIMER(n) GOSUB <**line number**>

PURPOSE:

Provides an event trap during real time.

REMARKS:

ON TIMER causes an event trap every (n) seconds. (n) must be a numeric expression from 1 to 86400 (1 second to 24 hours). Values outside this range generate an "Illegal function call" error.

The ON TIMER statement is executed only if a TIMER ON statement is executed to enable event trapping. If event trapping is enabled and the <line number> in the ON TIMER statement is not zero, VBASICA checks between statements to see if the time has been reached. If it has, a GOSUB is performed to the specified line.

If a TIMER OFF statement has been executed, the GOSUB is not performed and is not remembered.

If a TIMER STOP statement has been executed, the GOSUB is not performed, but will be performed as soon as a TIMER ON statement is executed.

When an event trap occurs (that is, the GOSUB is performed), an automatic TIMER STOP is executed so that recursive traps cannot occur. The RETURN from the trapping subroutine automatically performs a TIMER ON statement unless an explicit TIMER OFF was performed inside the subroutine.

You can use the RETURN <line number> form of the RETURN statement to return to a specific line number from the trapping subroutine. Use this type of return with care because any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active, and errors such as "FOR without NEXT" can result.

EXAMPLE:

The following example displays the time of day on line 1 every minute.

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
      .
      .
      .
10000 LET OLDROW=CSRLIN 'Save current Row
10010 LET OLDCOL=POS(0)'Save current Column
10020 LOCATE 1,1:PRINT TIMES$;
10030 LOCATE OLDROW,OLDCOL 'Restore Row & Col
10040 RETURN
```

Also see the TIMER ON, TIMER OFF, and TIMER STOP statements.

OPEN Statement

FORMAT:

OPEN [< dev >] < filespec > [FOR < mode >] AS [#] < file number > [LEN = < lrecl >]

PURPOSE:

Establishes addressability between a physical device and an I/O buffer in the data pool.

REMARKS:

< dev > is an optional part of the filename string.

< filespec > is a valid string expression for the file specification. Refer to Chapter 1.5 for more information. In the simplest case, it is a filename. The device may be specified separately in the < dev > field or as part of the < filespec >, or omitted entirely, in which case the default drive is assumed.

< mode > determines the initial positioning within the file and the action to be taken if the file does not exist. The valid modes and actions taken are the following:

INPUT	Position to the beginning of an existing file. The "File Not Found" error is given if the file does not exist.
OUTPUT	Position to the beginning of the file. If the file does not exist, one is created.
APPEND	Position to the end of the file. If the file does not exist, one is created.

If you omit the FOR < mode > clause, the initial position is at the beginning of the file. If VBASICA does not find the file, it creates one in the Random I/O mode. Records can be read or written randomly at any position within the file.

< file number > is an integer expression returning a number from 1 through 255. Use this number to associate an I/O buffer with a disk file or device. This association exists until you execute a CLOSE < file number > or CLOSE statement.

NOTE: At any time, you can have a particular file open under more than one file number. Therefore, you can use different modes for different purposes. Or, for program clarity, you can use different file numbers for different modes of access. Each file number has a different buffer, so you can keep several records from the same file in memory for quick access. However, you cannot open a file for sequential output or append if the file is already open.

< lrecl > is an integer expression from 2 to 32768. This value sets the record length used for random files; see the FIELD statement. If omitted, the record length defaults to 128-byte records.

When you OPEN FOR APPEND a disk file, the position is initially at the end of the file and the record number is set to the last record of the file. Then GET#(< lrecl >), LOF(< lrecl >), PRINT, WRITE, or PUT extend the file. The program can position elsewhere in the file with a GET statement. If this procedure is done, the mode is changed to random and the position moves to the record indicated.

After VBASICA moves the position from the end of the file, you can append records to the file by executing a GET #x,LOF(x)/ < lrecl > .

OPEN COM Statement

FORMAT:

OPEN < COM.specification > AS [#] < file number >

PURPOSE:

Allocates a buffer for I/O as OPEN for disk files.

REMARKS:

< COM.specification > is:

" < dev > :[< speed >][, < parity >][, < data >][, < stop >][,RS] [,CS < n >][,DS < n >][,CD < n >][,LF][,PE][,ASC or ,BIN]"

< dev > is a valid communications device. Valid devices are COM1: (port A) and COM2: (port B).

< speed > is a literal integer specifying the transmit/receive baud rate. Valid speeds are the following:

50, 75, 110, 150, 200, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 9600, 19200

< parity > is a one-character literal specifying the parity for transmit and receive as follows:

S	SPACE	Parity bit always transmitted and received as space (0 bit).
O	ODD	Odd transmit/receive parity checking.
M	MARK	Parity bit always transmitted and received as mark (1 bit).
E	EVEN	Even transmit/receive parity checking.
N	NONE	No transmit parity, no receive parity checking.

< data > is a literal integer indicating the number of transmit/receive data bits. The following are valid values:

5, 6, 7, 8

NOTE: 5 data bits with no parity is illegal. 8 data bits with any parity is illegal.

< stop > is a literal integer indicating the number of stop bits. The following are valid values:

1, 2

If omitted, then 75 and 110 bps transmit two stop bits; all others transmit one stop bit.

[RS] suppresses RTS (Request to Send).

[CS < n >] controls CTS (Clear to Send).

[DS < n >] controls DSR (Data Set Ready).

[CD < n >] controls CD (Carrier Detect).

[LF] sends a line feed character after each carriage return, including the carriage return sent as a result of the width setting. LF allows communications files to be printed on a serial line printer. Note that INPUT# and LINE INPUT# stop when they encounter a carriage return, ignoring the line feed, when used to read from a communications file opened with the LF option.

[PE] enables parity checking. The default is no parity checking. The PE option causes a device I/O error on parity errors and turns on the high order bit for 7 or less data bits. The PE option does not affect framing and overrun errors. These errors always turn on the high order bit and cause a device I/O error.

< file number > is an integer expression returning a valid file number. The number is associated with the file for as long as it is OPEN and refers other COM I/O statements to the file.

DEFAULTS: Missing parameters invoke the following defaults: speed = 300 bps, parity = EVEN, bits = 7.

NOTE: You can OPEN a COM device to only one file number at a time.

POSSIBLE ERRORS:

Any coding errors within the filename string result in the "Bad File Name" error. VBASICA gives no indication of the parameter in error.

The "Device Timeout" error occurs if VBASICA does not detect data set ready (DSR). Refer to hardware documentation or your MS-DOS manual for proper cabling instructions.

EXAMPLE:

VBASICA opens file 1 for communication with default values. Speed at 300 bps, even parity, 7 data bits, and 1 stop bit:

```
10 OPEN "COM1: " AS #1
```

VBASICA opens file 2 for communication at 2400 bps. Parity and number of data bits are the default values:

```
20 OPEN "COM1:2400 " AS #2
```

VBASICA opens file number 1 for Asynchronous I/O at 1200 bits/second, with no parity produced or checked, and sends and receives 8-bit bytes:

```
10 OPEN "COM1:1200,N,8" AS #1
```

OPTION BASE Statement

FORMAT:

```
OPTION BASE n
```

PURPOSE:

Declares the minimum value for array subscripts.

REMARKS:

n is 1 or 0.

The default base is 0.

EXAMPLE:

If the following statement is executed, the lowest value an array subscript can have is 1:

```
OPTION BASE 1
```

OUT Statement

FORMAT:

```
OUT I,J
```

PURPOSE:

Sends a byte to an output port.

REMARKS:

I and J are integer expressions in the range 0 to 65535.

The integer expression I is the port number, and the integer expression J is the data to be transmitted.

EXAMPLE:

This command:

```
100 OUT 12345,255
```

is the same as the following (in assembly language):

```
MOV   DX,12345
MOV   AL,255
OUT   DX,AL
```

PAINT Statement

FORMAT:

PAINT (< xstart >, < ystart >)[, < paint attribute >
[, < border attribute >]]

PURPOSE:

Fills an area on the screen with the selected color, for Graphics mode only.

REMARKS:

< xstart > and < ystart > are valid coordinates that specify the screen coordinates for the origin of painting.

< paint attribute > is the color VBASICA paints specified by a number from 0 to 3. It determines which color fills the area.

< border attribute > is the color of the edges of the figure VBASICA paints specified by a number from 0 to 3. Painting continues until VBASICA finds this color.

The PAINT statement fills in an arbitrary graphics figure with the specified paint color. If not specified, the paint attribute defaults to the foreground color (3 or 1), and the border attribute defaults to the paint attribute.

For example, you might want to fill in a circle of attribute 3 with attribute 0 (a black ball with a white border).

Only two attributes exist in high-resolution mode: whiting out an area until white is encountered, or blacking out an area until black is encountered.

PAINT must start on a nonborder point or it has no effect.

PAINT can fill any figure, but PAINTing jagged edges or very complex figures can result in the "Out of Memory" error. If you get this error, increase the amount of stack space available with the CLEAR statement.

VBASICA clips out-of-range coordinates.

EXAMPLE:

```
10 SCREEN 2
20 LINE (100,200)-(200,350),1,B
30 PAINT (150,225),1,1
```


Tiling

Tiling is the design of a PAINT pattern 8 bits wide and up to 64 bytes long. Each byte in the tile string masks 8 bits along the x-axis when putting down points. Construction of the tile mask works as follows. Use the syntax:

PAINT (x,y), CHR$(n)...CHR$(n)

where (n) is a number between 0 and 255 which will be represented in binary across the x-axis of the tile. Each CHR$(n) up to 64 generates an image of the bit arrangement of the code for that character. For example, the decimal number 85 is binary 01010101. The graphic image line on a black and white screen generated by CHR$(85) is an eight-pixel line, with even numbered pixels white, and odd pixels black. That is, each bit containing a 1 sets the associated pixel on and each bit filled with a 0 sets the associated bit off in monochrome mode. The ASCII character CHR$(85), U, is not displayed in this case.

If the current screen mode supports only two colors, the screen can be painted with X's with the following statement:

PAINT (320,100),CHR$(129) + CHR$(66) + CHR$(36) + CHR$(24) + CHR$(24) + CHR$(36) + CHR$(66 + CHR$(129)

This appears on the screen as:

x increases →

0,0	x							x	CHR$(129)	Tile byte 1
0,1		x					x		CHR$(66)	Tile byte 2
0,2			x			x			CHR$(36)	Tile byte 3
0,3				x	x				CHR$(24)	Tile byte 4
0,4				x	x				CHR$(24)	Tile byte 5
0,5			x			x			CHR$(36)	Tile byte 6
0,6		x					x		CHR$(66)	Tile byte 7
0,7	x							x	CHR$(129)	Tile byte 8

When supplied, < background attribute > specifies the background tile slice to skip when checking for boundary termination.

You cannot specify more than two consecutive bytes in the tile background slice that match the tile string. Specifying more than two results in an "Illegal function call" error.

EXAMPLE:

```
10 PAINT (5,15),2,0
```

begins painting at coordinates 5,15 with color 2 and border color 0, and fills to a border.

PLAY Statement

FORMAT:

PLAY < **string exp** >

PURPOSE:

Plays music as specified by < string exp > .

REMARKS:

< string exp > is a string expression returning a valid string conforming to the format described in Table 3-5.

PLAY implements a concept similar to DRAW by embedding a Music Macro Language into the string data type. The following table describes the single-character commands you can use with the PLAY statement.

Table 3-5: PLAY Commands

COMMAND	DESCRIPTION
A–G [#, +, −]	Plays the note. A # or + following the note indicates sharp, and − indicates flat.
L < n >	Length sets the length of each note. L4 is a quarter note, L1 is a whole note, and so on. n ranges from 1 to 64. The length can also follow the note when you want to change the length for only one note. In this case, A16 is equivalent to L16A.
MF	Music Foreground. Music created by PLAY or SOUND runs in the foreground. That is, VBASICA does not execute the next program statement until the last note, or rest, of subsequent PLAY statements are started.
MB	Music Background. Music created by SOUND or PLAY runs in the background. That is, each note or sound is placed in a buffer, allowing the VBASICA program to continue executing while music plays in the background. Up to 32 notes (or rests) can be played in the background at a time.

COMMAND	DESCRIPTION
MN	Music Normal. Each note plays 7/8ths of the time determined by L (length).
ML	Music Legato. Each note plays the full period set by L (length).
MS	Music Staccato. Each note plays 3/4ths of the time determined by L (length).
N <n>	Play note n. n can range from 0 to 84. In the seven possible octaves, 84 notes exist. N = 0 indicates rest.
> <n>	Go to the next higher octave and play note n. Each time note n is played, the octave increases until it reaches octave 6. For example, PLAY "> A" raises the octave and plays note A. Each time PLAY "> A" is executed, the octave goes up until it reaches octave 6; then each time PLAY "> A" executes, note A plays at octave 6.
< <n>	Go to the next lower n and play note n. Each time note n is played, the octave decreases, until it reaches octave 0. For example, PLAY "< A" lowers the octave and plays note A. Each time PLAY "< A" executes, the octave decreases until it reaches octave 0; then each time PLAY "< A"executes, note A plays at octave 0.
O <n>	Octave. Sets the current octave. Seven octaves (0..6) exist.
P <n>	Pause. P ranges from 1 to 64.
T <n>	Tempo. Sets the number of L4's in a second. n ranges from 32 to 255. The default is 120.
Period (.)	Period. A dot after a note plays the note 3/2 times the period determined by L (length) times T (tempo). Multiple dots can appear after the note. The period is scaled accordingly (for example, A. 3/2, A. . 9/4, A. . . 27/8). Dots can appear after a pause (P) and scale the pause length as described.
X <string>	Executes substring.

NOTE: Because of the slow clock interrupt rate, some notes do not play at higher tempos—for example, L64 at T255. Determine these note/tempo combinations through experimentation. In all forms of the PLAY command, the n argument can be a constant such as 12 or it can be the name of a variable.

To play tied notes, concatenate the expressions of the two notes. VBASICA plays the two notes continuously.

You can use X to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

EXAMPLE:

```
10 A$ = "BB-C"
20 B$ = "O4XA$;"
30 C$ = "L1CT50N3N4N5N6"
40 PLAY "P2XA$;XB$;XC$;"
```


PLAY(n) Function

FORMAT:

PLAY(n)

PURPOSE:

Returns the number of notes currently in the background music queue.

REMARKS:

(n) is a dummy argument and can be any value.

PLAY(n) returns 0 when you are in Music Foreground mode.

PLAY ON, PLAY OFF, and PLAY STOP Statements

FORMAT:

```
PLAY ON
PLAY OFF
PLAY STOP
```


PURPOSE:

PLAY ON enables PLAY event trapping, specified by the ON PLAY statement.

PLAY OFF disables PLAY event trapping.

PLAY STOP suspends PLAY event trapping.

REMARKS:

If a PLAY OFF statement was executed, the GOSUB is not performed and is not remembered.

If a PLAY STOP statement was executed, the GOSUB is not performed, but is performed as soon as a PLAY ON statement is executed.

When an event trap occurs (that is, the GOSUB is performed), VBASICA executes an automatic PLAY STOP so recursive traps cannot occur. The RETURN from the trapping subroutine automatically performs a PLAY ON statement unless an explicit PLAY OFF was performed inside the subroutine.

You can use the RETURN < line number > form of the RETURN statement to return to a specific line number from the trapping subroutine.

NOTE: Any other GOSUBs, WHILEs, or FORs active at the time of the trap remain active, and errors such as "FOR without NEXT" can result.

PMAP Function

FORMAT:

PMAP < **expression** > , < **function** >

PURPOSE:

Maps world coordinate expressions to physical locations or maps physical expressions to a world coordinate location for graphics mode.

< function > can be:

0 Maps world expression to physical x coordinate.

1 Maps world expression to physical y coordinate.

2 Maps physical expression to world x coordinate.

3 Maps physical expression to world y coordinate.

REMARKS:

The four PMAP functions allow you to find equivalent point locations between the world coordinates created with the WINDOW statement and the physical coordinate system of the screen or viewport as defined by the VIEW statement.

EXAMPLE:

If you define a WINDOW SCREEN (80,100)-(200,200) then the upper left coordinate of the window is (80,100) and the lower right is (200,200). The range of the screen coordinates can be (0,0) in the upper left corner and (639,199) in the lower right. Then,

```
X = PMAP(80,0)
```

returns the screen x coordinate of the window x coordinate 80:

```
0
```


The PMAP function in the statement:

```
Y = PMAP(200,1)
```

returns the screen y coordinate of the window y coordinate 200:

```
199
```

The PMAP function in the statement:

```
X = PMAP(619,2)
```

returns the "world" x coordinate that corresponds to the screen or viewport x coordinate 619:

```
199
```

The PMAP function in the statement:

```
Y = PMAP(100,3)
```

returns the "world" y coordinate that corresponds to the screen or viewport y coordinate 100:

```
140
```

POINT Function

FORMAT:

POINT (< xcoordinate > , < ycoordinate >)

or

POINT (< function >)

PURPOSE:

POINT (x,y) allows you to read the color number of a pixel from the screen. If the specified point is out of range, the value −1 is returned. In medium-resolution graphics, valid returns are 0 through 3, and in high-resolution, 0 through 1.

REMARKS:

< xcoordinate > and < ycoordinate > are the coordinates of the pixel to be referenced, for Graphics mode only.

POINT with one argument allows you to retrieve the current graphics cursor coordinates. Therefore:

a = POINT < function >

returns the value of the current x or y graphics accumulator, depending on the value of < function > , as follows:

< function >	= Action of POINT < function >
0	Returns the current physical x coordinate.
1	Returns the current physical y coordinate.
2	Returns the current logical x coordinate. If the WINDOW statement was not used, this returns the same value as the POINT(0) function.

3 Returns the current logical y coordinate if WINDOW is active, or else returns the current physical y coordinate, as in 1.

where the physical coordinate is the coordinate on the screen or current viewport.

EXAMPLE:

```
10 FOR I = 1 TO 400
20   IF POINT(I,I) <> 0 THEN GOTO 50   'a dot ?
30   PSET(I,I)       'put a dot if not one here
40   GOTO 60
50   PRESET(I,I)     'remove a dot if one here
60 NEXT I
```


POKE Statement

FORMAT:

POKE I,J

PURPOSE:

Writes a byte into a memory location.

REMARKS:

I and J are integer expressions.

I is the address of the memory location to be POKEd. I must be from 0 to 65536. J is the data to be POKEd. J must be from 0 to 255.

PEEK is the complementary function to POKE. The argument to PEEK is an address from which a byte is read.

EXAMPLE:

```
10 POKE &H5A00,&HFF
```

POS Function

FORMAT:

POS(X)

PURPOSE:

Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

See also the LPOS function.

EXAMPLE:

```
IF POS(X)>60 THEN PRINT CHR$(13)
```

PRESET Statement

FORMAT:

PRESET (<absolute x>,<absolute y>) [,<attribute>]

PRESET STEP (<x offset>,<y offset>) [,<attribute>]

PURPOSE:

3

Removes (turns off) or displays (turns on) one pixel from the screen, for Graphics mode only.

REMARKS:

PRESET has a syntax identical to PSET. The only difference is that if no third parameter is given for the background color, zero is selected. When you give a third argument, PRESET is identical to PSET.

If an out-of-range coordinate is given to PSET or PRESET, no action is taken, nor is an error given. If an attribute greater than 3 is given, the "Illegal function call" error results. Attribute value 2 is treated like 0 in high resolution and 3 is treated like 1 for compatibility with medium resolution.

VBASICA clips out-of-range coordinates.

EXAMPLE:

```
10 FOR I = 0 TO 100
20     PSET (I,I)
30 NEXT   'draw a diagonal line to (100,100)
40 FOR I = 100 TO 0 STEP -1
50 PRESET (I,I),0
60 NEXT   'remove the line just drawn
```

PRINT Statement

FORMAT:

PRINT [< expr list >]

PURPOSE:

Outputs data to the screen.

REMARKS:

If you omit < expr list >, VBASICA prints a blank line. If you include < expr list >, the values of the expressions appear on the screen. The expressions in the list can be numeric and/or string expressions. Strings must be enclosed by quotation marks.

Print Positions

VBASICA divides each line into print zones of 14 spaces. The punctuation that separates the items in the list determines the position of each printed item. A comma prints the next value at the beginning of the next zone. A semicolon prints the next value immediately after the last value. One or more spaces between expressions is equivalent to typing a semicolon.

If < expr list > ends with a comma or a semicolon, the next PRINT begins printing on the same line, spacing accordingly. If the < expr list > terminates without a comma or a semicolon, VBASICA prints a Return at the end of the line. If the printed line exceeds the screen width, VBASICA continues printing on the next physical line.

A space always follows printed numbers. A space precedes positive numbers. A minus sign precedes negative numbers.

If VBASICA can represent a single precision number with six or fewer digits in the unscaled format as accurately as represented in the scaled format, VBASICA outputs that number in the unscaled format. For example, 1E−7 is output as .0000001 and 1E−8 is output as 1E−08. If a double precision number can be represented with 16 or fewer digits in the unscaled format as accurately as represented in the scaled format, VBASICA outputs the number in the unscaled format. For example, 1D−16 is output as .0000000000000001 and 1D−17 is output as 1D−17.

EXAMPLE:

In the following example, the commas in the PRINT statement print each value at the beginning of the next print zone.

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
RUN
 10        0          -25        3125
Ok
```

In the following example, the semicolon at the end of line 20 prints both PRINT statements on the same line. Line 40 prints a blank line before the next prompt.

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
?9
9 SQUARED IS 81 AND 9 CUBED IS 729

?21
21 SQUARED IS 441 AND 21 CUBED IS 9261
```

In the following example, the semicolons in the PRINT statement print each value immediately after the preceding value. A space always follows a number, and a space precedes positive numbers.

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 PRINT J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```


PRINT USING Statement

FORMAT:

PRINT USING <**str expr**>;<**expr list**>

PURPOSE:

Prints strings or numbers in specified format.

REMARKS:

<expr list> is a list of string or numeric expressions to be printed, separated by semicolons.

<str expr> is a string literal or variable consisting of special formatting characters.

<expr list> and <str expr> determine the field size and the format of the printed strings.

String Fields

When PRINT USING is used to print strings, one of three formatting characters can be used to format the string field:

- The ! specifies that only the first character of the given string is printed.
- \n spaces\. Two characters from the string are to be printed. If you put spaces between the backslashes, each space adds another character to the printed string. For example, if you use one space, three characters are printed.
- If the string to be printed is longer than the field, VBASICA ignores the extra characters. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right. For example,

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!",A$;B$
40 PRINT USING "\ \";A$;B$
50 PRINT USING "\ \";A$;B$;"!!"
RUN
 LO
 LOOKOUT
 LOOK OUT !!
```

- The & specifies a variable-length string field. When you specify the field with the optional &, the string is output exactly as input. For example,

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!",A$;
30 PRINT USING "&";B$
RUN
 LOUT
```

Numeric Fields

When you print numbers with PRINT USING, VBASICA uses the following special characters to format the numeric field:

- A number sign (#) represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than the number of positions specified, the number is right-justified (preceded by spaces) in the field.
- You can insert a decimal point at any position in the field. If the format string specifies that a digit is to precede the decimal point, VBASICA always prints the digit (as 0 if necessary). VBASICA rounds numbers as necessary.


```
PRINT USING "##.##";.78
0.78

PRINT USING "###.##";987.654
987.65
```

In the following example, the three spaces inserted at the end of the format string separate the printed values on the line:

```
PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20   5.30   66.79   0.23
```

- A plus sign (+) at the beginning or end of the format string prints the sign of the number (plus or minus) before or after the number:

```
PRINT USING "+##.##";-68.95,2.4,55.6,-9
-68.95        +2.40   +55.60        -0.90
```

- A minus sign (−) at the end of the format field prints negative numbers with a trailing minus sign, as in this example:

```
PRINT USING "##.##-";-68.95,22.44900,-7.01
68.95-    22.45     7.01-
```

- A double asterisk (**) at the beginning of the format string fills the leading spaces in the numeric field with asterisks. The ** also specifies positions for two more digits. For example,

```
PRINT USING "**#.#";12.39,-0.9,765.1
*12.4     *-0.9     765.1
```

- A double dollar sign ($$) specifies two digit positions; one is the dollar sign. The $$ prints a dollar sign immediately to the left of the formatted number. Do not use the exponential format with $$. Do not use negative numbers unless the minus sign trails to the right. For example,


```
PRINT USING "$$###.##";456.78
$456.78
```

- The **$ specifies three digit positions, one of which is the dollar sign. A **$ at the beginning of a format string combines the effect of the preceding two symbols. Leading spaces are asterisk-filled. A dollar sign is printed before the number.

```
PRINT USING "**$##.##";2.34
**$2.34
```

- A comma specifies another digit position. A comma to the left of the decimal point in a formatting string prints a comma to the left of each third digit left of the decimal point. VBASICA prints a comma at the end of the format string as part of the string. The comma has no effect if used with the exponential format. For example,

```
PRINT USING "###,.###";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

- Four carets (^^^^) specify exponential format when located after digit-position characters. The four carets allow space for E, a plus sign, and two digits to be printed. Any decimal point position can be specified. The significant digits are left-justified, and the exponent is adjusted. Unless you specify a leading or trailing plus (or minus) sign, one digit position to the left of the decimal point is used to print a space or a minus sign.

```
PRINT USING "##.##";^^^^234.56
2.35E+02

PRINT USING ".####^^^^-";888888
.8889E+06

PRINT USING "+.##^^^^";123
.12E+03
```

- An underscore (_) causes the next character to be output as a literal character:

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

To print an underscore, put two underscore characters into the format string.

- A percent sign (%) prints a percent sign before a number if that number is larger than the specified numeric field. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error results.

PRINT# and PRINT# USING Statements

FORMAT:

PRINT# < filenum > ,[USING < str expr > ;] < expr list >

PURPOSE:

Writes data to a sequential disk file.

3

REMARKS:

< filenum > is the number used when the file was OPENed for output.

< str expr > consists of the formatting characters used with PRINT USING.

< expr list > is the numeric and/or string expressions written to the file.

PRINT# does not compress data on the disk. The data is written to the disk in the same form that VBASICA displays it on the screen by a PRINT statement. Delimit the data so it is input correctly to the disk.

Separate numeric expressions in < expr list > by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

If you use commas as delimiters, the extra blanks inserted between print fields are also written to disk.

Separate string expressions in < expr list > with semicolons. Use explicit delimiters to format the string expressions correctly on the disk.

For example, assume that A$ is "CAMERA" and B$ is "93604-1".

The statement:

```
PRINT#1,A$;B$
```

writes CAMERA 93604-1 onto disk. Because there are no delimiters, this line cannot be input as two separate strings. To correct the problem, insert delimiters into the PRINT# statement as shown:

```
PRINT#1,A$;",";B$
```

CAMERA,93604-1 is written to disk. In the new format, the data can be read back into two string variables.

If the strings contain commas, semicolons, significant leading blanks, Returns, or linefeeds, enclose each with explicit quotation marks by using CHR$(34). Assume that A$ is "CAMERA, AUTOMATIC" and B$ is "93604-1".

The statement:

```
PRINT#1,CHR$(34);A$,CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following to disk:

```
"CAMERA, AUTOMATIC" "93604-1"
```

and the statement:

```
INPUT#1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and "93604-1" to B$.

You can also use the PRINT# statement with the USING option to control the format of the disk file, as in this example:

```
PRINT#1,USING"$$###.##;";J;K;L
```

See Appendix F for more information on disk file formatting.

PSET Statement

FORMAT:

PSET (<absolute x>,<absolute y>) [,<attribute>]

PSET STEP (<x offset>,<y offset>) [,<attribute>]

PURPOSE:

Displays (turns on) or removes (turns off) one pixel from the screen, for Graphics mode only.

REMARKS:

<absolute x>, <absolute y>, <x offset>, and <y offset> are valid coordinates. See Chapter 1.7 for further information.

<attribute> is an expression returning a value 0 to 3, which determines the color of the point. An attribute of 0 sets the point to the background color, as described in Chapter 1.7.

If you omit the attribute argument, it defaults to 3 in Screen mode 1 and to a value of 1 in Screen mode 2.

VBASICA clips out-of-range coordinates.

EXAMPLE:

```
10 FOR I = 0 TO 100
20 PSET (I,I)
30 NEXT '(draw a diagonal line to (100,100))
40 FOR I = 100 TO 0 STEP -1
50 PSET (I,I),0
60 NEXT '(remove the line just drawn)
```

RANDOMIZE Statement

FORMAT:

RANDOMIZE [< expr >]

PURPOSE:

Reseeds the random number generator.

REMARKS:

< expr > is an integer, single or double precision expression used as the random number seed. If you omit < expr >, VBASICA suspends program execution and asks for a value. The following prompt appears on the screen:

```
Random Number Seed (-32768 to 32767)?
```

RANDOMIZE executes after you supply a value.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers, put a RANDOMIZE statement at the beginning of the program and change its argument each time you run the program.

EXAMPLE:

```
10 RANDOMIZE
20 FOR I=I TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767) ? 3
.88598  .484668  .586328  .119426  .709225
Ok

RUN
Random Number Seed (-32768 to 32767) 4
.803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3
.88598  .484668  .586328  .119426  .709225
Ok
```

READ Statement

FORMAT:

READ < **varlist** >

PURPOSE:

Reads values from a DATA statement and assigns them to variables.

REMARKS:

Always use a READ statement with a DATA statement. READ statements assign variables to DATA statement values, one for one. READ statement variables can be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" results.

A single READ statement can access one or more DATA statements (in order), or several READ statements can access the same DATA statement. If the number of variables in < varlist > exceeds the number of elements in the DATA statements, subsequent READ statements begin reading data at the first unread element. If no subsequent READ statements exist, VBASICA ignores the extra data.

Use the RESTORE statement to read DATA statements from the start.

EXAMPLE:

The following program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
      .
      .
      .
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
      .
      .
      .
```

The following program READs string and numeric data from the DATA statement in line 30:

```
LIST
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
 CITY        STATE     ZIP
 DENVER,     COLORADO  80211
Ok
```

REM Statement

FORMAT:

REM < **remark** >

PURPOSE:

Inserts explanatory remarks and comments into a program.

REMARKS:

VBASICA does not execute REM statements. They are output exactly as entered when the program is listed.

REM statements can be branched into from a GOTO or GOSUB statement. Execution continues at the first executable statement after the REM statement.

You can add remarks to the end of a line by preceding the remark with a single quotation mark.

WARNING: Do not use REM in a DATA statement. VBASICA treats it as data, not as a remark.

EXAMPLE:

```
      .
      .
      .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM+V(I)
      .
      .
      .
      .
120 FOR I=1 TO 20  'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
      .
      .
      .
```


RENUM Command

FORMAT:

RENUM [[< new number >][,[< old number >][, < increment >]]]

PURPOSE:

Renumbers program lines.

REMARKS:

< new number > is the first line number in the new sequence. The default is 10.

< old number > is the line where renumbering is to begin. The default is the first line of the program.

< increment > is the increment used in the new sequence. The default is 10.

RENUM changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, VBASICA prints the error message "undefined line xxxxx in yyyyy". RENUM does not change the incorrect line number reference (xxxxx), but the line number (yyyyy) can be changed.

You cannot use RENUM to change the order of program lines or to create line numbers greater than 65529. An "Illegal function call" error results in both cases.

EXAMPLES:

In the following example, the statement renumbers the entire program. The first new line number is 10. Lines increment by 10.

```
RENUM
```

In the following statement, VBASICA renumbers the entire program. The first new line number is 300. Lines increment by 50.

```
RENUM 300,,50
```

In the following statement, VBASICA renumbers the lines beginning at 900. The new numbering begins with 1000. Each line increments by 20.

```
RENUM 1000,900,20
```

RESET Command

FORMAT:

RESET

PURPOSE:

Closes all files.

REMARKS:

RESET closes all open files and writes all blocks in memory to disk. You must close all files before you remove a disk from its drive.

EXAMPLE:

```
998 RESET
999 END
```

RESTORE Statement

FORMAT:

RESTORE [< line >]

PURPOSE:

Reads DATA statements, beginning at a specified line.

REMARKS:

< line > is the number of a program line.

After VBASICA executes a RESTORE statement, the next READ statement begins with the first item in the program's first DATA statement. If you specify < line >, the next READ statement starts at the first item in the specified DATA statement.

EXAMPLE:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57,68,79
    .
    .
    .
```

RESUME Statement

FORMAT:

RESUME

RESUME 0

RESUME NEXT

RESUME < **line** >

3

PURPOSE:

Continues program execution after an error-recovery procedure.

REMARKS:

< line > is the number of a program line.

You can use any of the four formats shown, depending on where execution resumes:

- RESUME and RESUME 0 resume execution at the statement that caused the error.
- RESUME NEXT resumes execution at the statement immediately following the one that caused the error.
- RESUME < line > resumes execution at the line specified.

A RESUME statement that appears outside of an error-trapping routine causes a "RESUME without error" message to appear.

EXAMPLE:

```
10 ON ERROR GOTO 900

   .
   .

900 IF (ERR=230)AND(ERL=90) THEN PRINT
    "TRY AGAIN":RESUME 80
   .
   .
```


RETURN Statement

FORMAT:

RETURN [< line number >]

PURPOSE:

Returns from a GOSUB.

REMARKS:

The line number is intended for use with event trapping. The event trap routine might want to go back into the VBASICA program at a fixed line number while still eliminating the GOSUB entry the trap created.

Use the nonlocal return with care. Any other GOSUB, WHILE, or FOR active at the time of the trap remains active. If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine results in the "NEXT without FOR" error.

EXAMPLE:

```
100 RETURN 900
```

RMDIR Command

FORMAT:

RMDIR < **pathname** >

PURPOSE:

Removes an existing directory.

3

REMARKS:

< pathname > is the name of the directory to be deleted. RMDIR works exactly like the MS-DOS command RMDIR. < pathname > must be a string of less than 63 characters.

The < pathname > to be removed must be empty of any files except the working directory (.) and the parent directory (..) or else VBASICA gives a "Path not found" or a "Path/File access" error.

EXAMPLE:

In the following statement, the SALES directory on the current drive is removed:

```
RMDIR "\SALES"
```

RND Function

FORMAT:

RND[(X)]

PURPOSE:

Returns a random number between 0 and 1. VBASICA generates the same sequence of random numbers each time the program is RUN unless you reseed the random number generator.

If X is less than zero, RND always restarts the same sequence for any given X. If X is greater than zero, or X is omitted, RND generates the next random number in the sequence. If X equals zero, RND repeats the last number generated.

EXAMPLE:

The following example prints five random numbers between 0 and 100.

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT I
```

RUN Command

FORMAT 1:

RUN [<line>]

FORMAT 2:

RUN <filespec> [,R]

3

PURPOSE:

Format 1 executes the program in memory. Format 2 loads a disk file into memory and runs it.

REMARKS:

<line> is the number of a program line.

With Format 1, execution begins at <line>. If you do not specify <line>, execution starts at the lowest line number. VBASICA returns to command level after it executes a RUN.

In Format 2, <filespec> is a string expression for the specification. Refer to Chapter 1.5 for more information on file specifications.

RUN closes all open files and deletes the current contents of memory before loading the designated program. If you use the R option, all data files remain open.

EXAMPLE:

```
RUN "NEWFIL",R
```

VBASICA supports the RUN and RUN <line number> forms of the RUN statement; however, it does not support the R option. If you want this feature, use the CHAIN statement.

SAVE Command

FORMAT:

SAVE < **filespec** > **[,A,P]**

PURPOSE:

Saves a VBASICA program file on disk or another device.

REMARKS:

< filespec > is a string expression representing the file specification. Refer to Chapter 1.5 for more information on file specifications. Device specifications other than the diskette are legal.

The A option saves the file in ASCII format. If the A option is not specified, VBASICA saves the file in a compressed binary format. ASCII format takes more space on the disk, but some actions require that files be in ASCII format. For example, the MERGE command requires an ASCII format file, and some operating system commands such as TYPE can require an ASCII format file.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

EXAMPLE:

```
SAVE "COM2",A
```

Saves the program COM2.BAS in ASCII format.

```
SAVE "PROG",P
```

Saves the program PROG.BAS as a protected file which cannot be altered.

SCREEN Statement

FORMAT:

SCREEN [< mode >] [,[< burst >] [,[< apage >] [, < vpage >]]]

PURPOSE:

Sets the screen attributes.

REMARKS:

< mode > is a valid numeric expression returning an unsigned integer value. Valid modes are the following:

0	Alpha mode at current width (40 or 80)
1 and 2	Graphics modes

< burst > is a valid numeric expression returning a Boolean result.

< apage > is the active page. This value can be nonzero only on Screen 0. In this mode, it is a numeric expression returning an unsigned integer from 0 to 7 for width 40, or 0 to 3 for width 80. VBASICA sends output to the screen to this page.

< vpage > is the visible page. It follows the same rules as < apage >, but selects the page to be displayed on the screen. If < vpage > is omitted, the visible page is set to the active page.

If all parameters are legal and < mode > or < burst > changes from their previous values, VBASICA clears the screen. In that case, the background color is reset to black, and the foreground color to white.

If the mode is 0, and you specify only <apage> and <vpage>, VBASICA changes display pages for viewing. Initially, both active and visual pages default to zero. By manipulating active and visual pages, you can display one page while constructing another. You can switch both pages instantaneously.

NOTE: Only one cursor is shared between the pages. If you are going to switch active pages back and forth, save the cursor position on the current page (using POS(0) and CSRLIN) before changing to another active page. When you return to the original page, you can return the cursor to the saved position using the LOCATE statement.

RULES:

1. Any values you enter outside these ranges result in the "Illegal function call" error. VBASICA retains the previous values.
2. You can omit any parameter. Omitted parameters assume the old value.

EXAMPLE:

```
10 SCREEN 0,0,0      'select alpha mode
10 SCREEN 2          'select hi-res graphics
```

SCREEN Function

FORMAT:

x = SCREEN (<row>,<col> [,<boolean>])

PURPOSE:

Returns the ASCII code value of the character from the screen at the specified row (line) and column.

3

ACTION:

x is a numeric variable receiving the ordinal returned.

<row> is a valid numeric expression returning an unsigned integer in the range 1 to 25.

<col> is a valid numeric expression returning an unsigned integer in the range 1 to 40 or 1 to 80, depending on the width.

<boolean> is a valid numeric value or expression giving a true or false (Boolean) result.

The ASCII code value of the character at the specified coordinates is stored in the numeric variable. If the optional parameter <boolean> is given and nonzero, the color attribute for the character is returned instead. The color attribute is a number in the range 0 to 255. This parameter may be interpreted as follows:

- (<param> MOD 16) is the foreground color. (For the standard screen and printer, values of 8 through 15 indicate high intensity for "colors" 0 through 7.)
- ((<param> MOD 128) − foreground) is the background color, where foreground is calculated as shown.
- (<param> > 127) is true (−1) if the character is blinking, false (0) if not.

RULE: Any values entered outside these ranges result in the "Illegal function call" error.

EXAMPLE:

```
100 X = SCREEN (10,10)   'Returns 65 if the
        'character at 10,10 is an 'A'.
110 X = SCREEN (1,1,1)   'Returns the color
        'attribute of the character in the
        'upper left corner of the screen.
```


SGN Function

FORMAT:

SGN(X)

PURPOSE:

Indicates the value of X, relative to zero:

- If X > 0, SGN(X) returns 1.
- If X = 0, SGN(X) returns 0.
- If X < 0, SGN(X) returns − 1.

EXAMPLE:

```
ON SGN(X)+2 GOTO 100,200,300
```

Branches to 100 if X is negative, 200 if X is 0, and 300 if X is positive.

SHELL Statement

FORMAT:

SHELL [< command-string >]

PURPOSE:

Exits the VBASICA program, runs a .COM, .EXE, or .BAT program or a built-in MS-DOS function such as DIR or TYPE, and returns to the VBASICA program at the line after the SHELL statement.

REMARKS:

A .COM, .EXE, or .BAT program or an MS-DOS function that runs under the SHELL statement is called a child process. SHELL executes child processes by loading and running a copy of COMMAND with the /C switch. By using COMMAND in this way, command line parameters are passed to the child. Standard input and output can be redirected, and built-in commands such as DIR, PATH, and SORT can be executed.

The < command-string > must be a valid string expression containing the name of a program to run and optional command arguments.

The program name in < command-string > can have any extension. If you supply no extension, COMMAND looks for a .COM file, then an .EXE file, and finally, a .BAT file. If COMMAND is not found, SHELL issues a "File not found" error. VBASICA generates no error if COMMAND cannot find the file specified in < command-string > .

COMMAND processes any text separated from the program name by at least one blank as program parameters.

VBASICA remains in memory while the child process is running. When the child finishes, VBASICA continues.

WARNING: Do not attempt to SHELL to VBASICA as a child process. This option is not supported, and causes unpredictable results. Integrity of the parent VBASICA might or might not be harmed (in RAM only).

SHELL with no <command-string> gives you a new COMMAND shell. You can now do anything that COMMAND allows. When you are ready to return to VBASICA, enter the MS-DOS command EXIT.

EXAMPLES:

```
SHELL (get a new COMMAND)
A>DIR (you type DIR to see files)
A>EXIT (you type EXIT to return to VBASICA)
Ok (now you are back in VBASICA)
```

The following example writes some data to sort, uses SHELL to sort it, then reads the sorted data to write a report:

```
900 OPEN "SORTIN.DAT" FOR OUTPUT AS 1
950 REM ** write data to be sorted
1000 CLOSE 1
1010 SHELL "SORT SORTIN.DAT SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS 1
1030 REM ** Process the sorted data
```

```
10 SHELL "DIR | SORT > FILES."
20 OPEN "FILES." FOR INPUT AS 1
```

SIN Function

FORMAT:

SIN(X)

PURPOSE:

Returns the sine of X, where X is in radians.

REMARKS:

COS(X) = SIN(X + 3.14159/2).

EXAMPLE:

```
PRINT SIN(1.5)
```

yields

```
.9974951
```

See also the COS function.

SOUND Statement

FORMAT:

SOUND < **frequency** > , < **duration** >

PURPOSE:

Generates a sound from the speaker of a specified frequency for a specified duration.

REMARKS:

< frequency > is the desired frequency in Hertz. It is a valid numeric expression returning an unsigned integer from 37 to 32767.

< duration > is the desired duration in clock ticks. It is a valid numeric expression returning an unsigned integer from 0 to 65535. Currently, clock ticks occur 18.2 times per second. If the duration is zero, any currently playing sound is turned off; if no sound is playing there is no effect.

Sounds can be buffered to prevent execution from stopping when VBASICA encounters a new SOUND statement. See the MB command in PLAY.

EXAMPLE:

```
2500 SOUND RND*1000+37,2    'creates random sounds.
```

SPACE$ Function

FORMAT:

SPACE$(I)

PURPOSE:

Returns a string of spaces of length I.

REMARKS:

The expression I is rounded to an integer and must be from 0 to 255.

EXAMPLE:

```
10 FOR I=1 TO 5
20 X$=SPACE$(I)
30 PRINT X$;I
40 NEXT I
```

yields

```
1
 2
  3
   4
    5
```

Also see the SPC function.

SPC Function

FORMAT:

SPC(n)

PURPOSE:

Skips spaces in a PRINT statement. n is the number of spaces skipped.

REMARKS:

You can only use SPC with PRINT and LPRINT statements. n must be from 0 to 255. A semicolon (;) is assumed to follow the SPC(n) command.

EXAMPLE:

```
PRINT "OVER" SPC(15) "THERE"
```

yields

```
OVER                THERE
```

Also see the SPACE$ function.

SQR Function

FORMAT:

SQR(X)

PURPOSE:

Returns the square root of X.

REMARKS:

X must be greater than or equal to 0.

EXAMPLE:

```
10 FOR X = 10 to 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT X
```

yields

```
10  3.162278
15  3.872984
20  4.472136
25  5
```

STOP Statement

FORMAT:

STOP

PURPOSE:

Terminates program execution and returns to command level.

REMARKS:

You can use STOP statements anywhere in a program to terminate execution. STOP is often used for debugging. When VBASICA encounters a STOP, the following message is printed:

```
Break in line nnnnn
```

The STOP statement doesn't close files.

VBASICA always returns to command level after a STOP is executed. Resume execution by issuing a CONT command.

EXAMPLE:

```
10 INPUT A,B,C
20 K = A^2*5.3:L = B^3/.26
30 STOP
40 M = C*K + 100:PRINT M
```

yields

```
? 1,2,3        (you enter 1,2,3)
BREAK IN 30
Ok

PRINT L        (you enter PRINT L)
30.76923
Ok
CONT           (you enter CONT)
115.9
```

STR$ Function

FORMAT:

STR$(n)

PURPOSE:

Returns a string representation of the value of n.

EXAMPLE:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB
30 100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

STRING$ Function

FORMAT:

STRING$(I,J)

STRING$(I,X$)

PURPOSE:

Returns a string of length I whose characters all have ASCII code J or the first character of X$.

EXAMPLE:

This example:

```
10 DASH$ = STRING$(7,45)
20 PRINT DASH$;"MONTHLY REPORT";DASH$
```

yields

```
-------MONTHLY REPORT-------
```

This example:

```
10 LET A$ = "HOUSTON"
20 LET X$ = STRING$(8,A$)
30 PRINT X$
```

yields

```
HHHHHHHH
```

SWAP Statement

FORMAT:

SWAP <**var1**>,<**var2**>

PURPOSE:

Exchanges the values of two variables.

3

REMARKS:

Any type variable (integer, single precision, double precision, string) can be SWAPped. Both variables must be of the same type; otherwise, a "Type mismatch" error results.

EXAMPLE:

```
LIST
10 A$= "ONE" :B$= "ALL" : C$= "FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

SYSTEM Command

FORMAT:

```
SYSTEM
```

PURPOSE:

Leaves VBASICA and returns to MS-DOS.

REMARKS:

All files are closed before exiting VBASICA.

TAB Function

FORMAT:

```
TAB(I)
```

PURPOSE:

Moves the print position to I.

REMARKS:

If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be from 1 to 255. You can use TAB only in PRINT and LPRINT statements.

EXAMPLE:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$, B$
30 PRINT A$ TAB(25) B$
40 DATA "G.T. JONES", "$25.00"
```

yields

```
NAME          AMOUNT
G.T. JONES    $25.00
```


TAN Function

FORMAT:

TAN(X)

PURPOSE:

Returns the tangent of X, where X is in radians.

REMARKS:

If TAN overflows, VBASICA displays the "Overflow" error message. VBASICA supplies machine infinity with the appropriate sign as the result, and execution continues.

EXAMPLE:

```
10Y = Q*TAN(X)/2
```

TIME$ Function and Statement

FORMAT:

TIME$ = < **string expr** >	Sets the current time.
< **string expr** > = **TIME$**	Gets the current time.

PURPOSE:

Sets or retrieves the current time.

REMARKS:

< string expr > is a valid string literal or variable.

VBASICA fetches and assigns the current time to the string variable if TIME$ is the expression in a LET or PRINT statement.

VBASICA stores the current time if TIME$ is the target of a string assignment.

RULES:

1. If < string expr > is not a valid string, the "Type mismatch" error results.
2. For < string var > = TIME$, TIME$ returns an 8-character string in the form "hh:mm:ss" where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).
3. For TIME$ = < string expr >, < string expr > can take one of the following forms:
   a. hh sets the hour. Minutes and seconds default to 00.
   b. hh:mm sets the hour and minutes. Seconds default to 00.
   c. hh:mm:ss sets the hour, minutes, and seconds.

You can omit a leading zero from any of these values, but you must include at least one digit. For example, if you want to set the time as one half hour after midnight, you can enter TIME$ = "0:30", but not TIME$ = ":30".

If any of the values are out of range, VBASICA issues the "Illegal function call" error. VBASICA retains the previous time.

EXAMPLE:

```
TIME$ = "08:00"
Ok
PRINT TIME$
08:00:04
Ok
```

This program displays the current date and time on the 25th line of the screen and chimes on the hour:

```
10 KEY OFF:SCREEN 0:WIDTH 40:CLS
20 LOCATE 25,5
30 PRINT DATE$,,TIME$
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 0 THEN 1010
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20
1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4 :GOTO 20
```

TIMER ON, TIMER OFF, and TIMER STOP Statements

FORMAT:

```
TIMER ON
TIMER OFF
TIMER STOP
```

PURPOSE:

TIMER ON enables event trapping during real time.

TIMER OFF disables event trapping during real time.

TIMER STOP suspends real-time event trapping.

REMARKS:

The TIMER ON statement enables real time event trapping by an ON TIMER statement (see ON TIMER Statement). While ON TIMER enables trapping, VBASICA checks between every statement to see if the timer has reached the specified level. If it has, VBASICA executes the ON TIMER statement.

TIMER OFF disables the event trap. If an event occurs it is not remembered if you use a subsequent TIMER ON.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an ON TIMER statement is executed as soon as you enable trapping.

Also see the ON TIMER statement.

TRON/TROFF Commands

FORMAT:

TRON

TROFF

PURPOSE:

Traces the execution of program statements.

REMARKS:

Use TRON as an aid to debugging. The TRON statement (executed in direct or indirect mode) enables a trace flag that prints each line number of the program as that line is executed. The numbers are enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

EXAMPLE:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
 [10][20][30][40] 1  10   20
 [50][60][30][40] 2  20   30
 [50][60][70]
Ok
TROFF
Ok
```

USR Function

FORMAT:

USR [< digit >] [(< argument >)]

PURPOSE:

Calls an assembly language subroutine.

REMARKS:

< digit > specifies the USR routine being called. See the DEF USR statement for rules governing < digit >. If you omit < digit >, VBASICA assumes USR0.

< argument > is the value passed to the subroutine. It may be any numeric or string expression. If a segment other than the default segment (data segment) is to be used, a DEF SEG statement must be executed prior to a USR function call. The address given in the DEF SEG statement determines the segment address of the subroutine.

For each USR function, a corresponding DEF USR statement must be executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

EXAMPLE:

```
100 DEF SEG = &H8000
110 DEF USR0=0
120 X=5
130 Y = USR0 (X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the value must be consistent with the argument passed.

VAL Function

FORMAT:

VAL(< string >)

PURPOSE:

Returns the numeric value of < string > . The VAL function strips leading blanks, tabs, and linefeeds from the argument string. For example,


```
VAL("-3")
```

returns − 3.

REMARKS:

< string > must be a numeric value stored as a string.

VAL stops scanning the string for numeric characters as soon as it encounters either:

- Any nonnumeric character other than blank, tab, or linefeed.
- Any nonnumeric character (including blank, tab, or linefeed) after having found any numeric character(s).

If < string > contains no numeric characters, VAL returns 0 (zero).

See the STR$ function for details on numeric-to-string conversion.

EXAMPLES:

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL (ZIP$) <90000 OR VAL (ZIP$)>96699
   THEN PRINT NAME$ TAB (25) "OUT OF STATE"
30 IF VAL (ZIP$)>=90801 AND VAL (ZIP$)<=90815
   THEN PRINT NAME$ TAB (25) "LONG BEACH"
    .
    .
```

The following example:

```
1000 ADDRESS$="27 So. Spring St."
1010 NUMBER=VAL(ADDRESS$)
1020 PRINT NUMBER
```

yields

```
27
```

This example yields 0:

```
PRINT VAL("ABC")
```


VARPTR Function

FORMAT:

x = VARPTR(< variable >)

y = VARPTR([#] < file number >)

PURPOSE:

For variables, returns the location in memory of the variable. For files, the VARPTR function returns the address of the first byte of the file control block (FCB) for the opened file.

REMARKS:

For both formats, the address returned is an integer from 0 to 65536. This number is the offset into the current segment of memory as defined by the DEF SEG statement.

The first format returns the address of the first byte of data identified with <variable>. Assign a value to <variable> prior to the VARPTR call, or you get an "Illegal function call" error. You may use any type variable (numeric, string, array element).

NOTE: Assign all simple variables before calling VARPTR for an array, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to a machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

The second format returns the starting address of the file control block for the specified file. This is not the same as the MS-DOS file control block. The file must be OPENed before the call to VARPTR.

<file number> is tied to a currently open file. Offsets to information in the FCB from the address returned by VARPTR are:

OFFSET	SIZE	CONTENTS	
0	1	Mode	The mode in which the file was opened: 1 Input Only 2 Output Only 4 Random I/O 16 Append Only 32 Internal Use 64 Future Use 128 Internal Use
1	38	FCB	Disk File Control Block.
39	2	CURLOC	Number of sectors read or written for sequential access. For Random access, it contains the last record number + 1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in input buffer.
43	3	***	Reserved for future expansion.

OFFSET	SIZE	CONTENTS	
46	1	DEVICE	Device number: 0–9 Disks A: through J: 255 KYBD: 254 SCRN: 253 LPT1: 251 COM1: 250 COM2: 249 LPT2: 248 LPT3:
47	1	WIDTH	Device width.
48	1	POS	Position in buffer for PRINT.
49	1	FLAGS	Internal use during LOAD/SAVE. Not used for data files.
50	1	OUTPOS	Output position used during tab expansion.
51	128	BUFFER	Physical data buffer. Used to transfer data between MS-DOS and VBASICA. Use this offset to examine data in sequential I/O mode.
179	2	VRECL	Variable-length record size. Default is 128. Set by length option in OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk file only. Output position for PRINT, INPUT, and WRITE.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine file data in Random I/O mode.

EXAMPLE:

```
10 OPEN "DATA.FIL" AS #1
20 FCBADR = VARPTR(#1) 'FCBADR contains start of FCB
30 DATADR = FCBADR+188 'DATADR contains address of
   'data buffer.
40 A$ = PEEK (DATADR) 'A$ contains lst byte in
   'data buffer.
```

VARPTR$ Function

FORMAT:

VARPTR$ (<variable name>)

PURPOSE:

Returns a character form of the variable's memory address. The form is compatible for programs that might be compiled later.

REMARKS:

<variable name> is the name of a variable in the program.

VARPTR$ executes substrings with the DRAW and PLAY statements in programs that will later be compiled. With programs that will not be later compiled, the standard syntax of the PLAY and DRAW statements is sufficient to produce desired effects.

You must assign a value to the variable before calling VARPTR$. Otherwise, an "Illegal function call" error results. Variables are defined by executing any reference to the variable.

VARPTR$ returns a three-byte string in the form:

byte 0 = type
byte 1 = low byte of address
byte 2 = high byte of address

The individual parts of the string are not considered characters.

NOTE: Because array addresses, string addresses and file data blocks change whenever a new variable is assigned, it is unsafe to save the result of a VARPTR function in a variable. Execute VARPTR before each use of the result.

EXAMPLE:

```
10 PLAY "X" + VARPTR$(A$)
```

Uses the subcommand X (execute), plus the contents of A$, as the string expression in the PLAY statement.

VIEW Statement

FORMAT:

VIEW [[SCREEN] [(Vx1,Vy1)-(Vx2,Vy2) [,[< color >][,[< border >]]]]]

PURPOSE:

Defines the screen limits for graphics activity, in Graphics mode only.

REMARKS:

VIEW defines a "Physical Viewport" limit from Vx1, Vy1 (upper left x,y coordinates) to Vx2, Vy2 (lower right x,y coordinates). The x and y coordinates must be within the physical bounds of the screen. The physical viewport defines the rectangle within the screen into which you can map graphics.

RUN, and RUN, SCREEN, and VIEW with no arguments, define the entire screen as the viewport.

With the < color > attribute, you can fill the view area with a color. If you omit color, VBASICA does not fill the view area.

With the < border > attribute, you can draw a line surrounding the viewport if space for a border is available. If you omit < border >, VBASICA draws no border.

The [SCREEN] option dictates that the x and y coordinates are absolute to the screen, not relative to the border of the physical viewport, and VBASICA plots only graphics within the viewport.

Out-of-range coordinates are clipped.

EXAMPLE:

For the following form, all points plotted are relative to the viewport. That is, Vx1 and Vy1 are added to the x and y coordinates before plotting the point on the screen.


```
VIEW (Vx1,Vy1)-(Vx2,Vy2)
```

If the following command is executed, then the point set down by the statement PSET (0,0),3 will be at the physical screen location 10,10.

```
VIEW (10,10)-(200,100)
```

For the following form, all coordinates are screen absolute rather than viewport relative:

```
VIEW SCREEN (Vx1,Vy1)-(Vx2,Vy2)
```

If VBASICA executes the following, then the point set down by the statement PSET (0,0),3 will not appear because 0,0 is outside the viewport. PSET (10,10),3 is within the viewport, and places the point in the upper left corner of the viewport.

```
VIEW SCREEN (10,10)-(200,100)
```

Many VIEW statements can be executed. If the newly described viewport is not wholly within the previous viewport, the screen can be reinitialized with the VIEW statement. Then the new viewport can be stated. If the new viewport is entirely within the previous one, as in the following example, the intermediate VIEW statement isn't necessary.

This example opens three viewports, each smaller than the previous one. In each case, a line defined to go beyond the borders is programmed, but appears only within the viewport border.

```
280 VIEW:REM ** Make the viewport the entire screen
300 VIEW (10,10) - (300,180),,1
320 CLS
340 LINE (0,0) -(310,190),1
360 LOCATE 1,11: PRINT "A big viewport"
380 VIEW SCREEN (50,50)-(250,150),,1
400 CLS:REM**Note, CLS clears only viewport
420 LINE (300,0)-(0,199),1
440 LOCATE 9,9: PRINT "A medium viewport"
460 VIEW SCREEN (80,80)-(200,125),,1
480 CLS
500 CIRCLE (150,100),20,1
520 LOCATE 11,9: PRINT "A small viewport"
```

VIEW PRINT Statement

FORMAT:

VIEW PRINT [< top screen line > TO < bottom screen line >]

PURPOSE:

Sets the boundaries of the screen text window.

3

REMARKS:

VIEW PRINT without top and bottom line parameters initializes the whole screen area as the text window.

Statements and functions that operate within the defined text window include CLS, LOCATE, PRINT, and SCREEN.

The Screen Editor limits functions such as scroll and cursor movement to the text window.

Also see the VIEW statement.

WAIT Statement

FORMAT:

WAIT < **port** > , < **and byte** > [, < **xor byte** >]

PURPOSE:

Suspends program execution while monitoring the status of a machine port.

REMARKS:

< port > is a numeric expression returning an integer from 0 to 65535.

< and byte > is a numeric expression returning an integer from 0 to 255 and matches a byte coming in from the < port > .

< xor byte > is a numeric expression returning an integer from 0 to 255 and checks a byte coming in from the < port > .

The WAIT statement suspends execution until a specified machine input port develops a specified bit pattern. The data read at the port is XOR'ed with the integer expression XOR byte and then AND'ed with the AND byte. If the result is zero, VBASICA loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement.

WARNING: It is possible to enter an infinite loop with the WAIT statement. CTRL-C exits the loop.

WHILE...WEND Statement

FORMAT:

```
WHILE < expr >
      .
      .
[ < loop  statements > ]
      .
      .
WEND
```


PURPOSE:

Executes a series of statements in a loop as long as a given condition is true.

REMARKS:

If < expr > is not zero (that is, true), the loop statements are executed until the WEND statement is encountered. VBASICA then returns to the WHILE statement and checks < expr >. If < expr > is still true, the process is repeated. If < expr > is not true, execution resumes at the statement after the WEND statement. WHILE/WEND loops can be nested to any level. Each WEND matches the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error; an unmatched WEND statement causes a "WEND without WHILE" error.

EXAMPLE:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
                SWAP A$(I), A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

WIDTH Statement

FORMAT:

WIDTH < **size** >

WIDTH < **file no.** > , < **size** >

WIDTH < **dev** > , < **size** >

PURPOSE:

Sets the printed line width in number of characters for the screen and line printer.

REMARKS:

< size > is a valid numeric expression returning an integer result from 0 to 255. This value is the new width.

< file no. > is a valid numeric expression returning an integer. This value is the number of the file OPENed.

< dev > is a valid string expression returning the device identifier.

Depending on the device specified, the following actions are possible:

```
WIDTH <size>
WIDTH "SCRN:", <size>
```

These commands set the screen width. VBASICA allows only 40- or 80-column width.

NOTE: Changing the screen width clears the screen. Screen mode 1 is always 40 columns wide, and Screen mode 2 is always 80 columns. If you are in either of these modes and ask to change the width, the mode changes to the appropriate one of these two modes.

```
WIDTH "LPT1:",<size>
```

Used as a deferred width assignment for the line printer. This form of WIDTH stores the new width value without changing the current width setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS < number > uses this value for width while the file is open.


```
WIDTH <file no.>,<size>
```

If the file is open to LPT1:, the line printer's width is immediately changed to the new size specified. This allows the width to be changed while the file is open. This form of WIDTH affects only LPT1:.

RULES:

1. Valid widths for the screen are 40 and 80. The valid width for the line printer is 1 to 255. Any value entered outside these ranges results in the "Illegal function call" error. VBASICA retains the previous value.
2. Width has no effect on the keyboard (KYBD:).
3. Maximum Epson line printer width is 80. WIDTH, however, does not complain about values between 80 and 255.
4. Specifying WIDTH 255 for the line printer (LPT1:) disables line folding.

EXAMPLE:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
        .
        .
        .
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a line printer width of 75 characters per line. Line 20 opens file #1 to the line printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current line printer width to 40 characters per line.

WINDOW Statement

FORMAT:

WINDOW [[SCREEN] (Wx1,Wy1)-(Wx2,Wy2)]

PURPOSE:

Defines the logical dimensions of the current viewport, for Graphics mode only.

REMARKS:

(Wx1,Wy1)-(Wx2,Wy2) are the world coordinates you specify to define the coordinates of the lower left and upper right screen border.

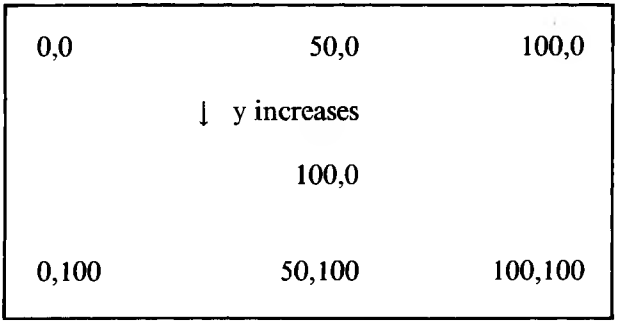
SCREEN inverts the y-axis of the world coordinates so that screen coordinates follow the traditional Cartesian system: x increases left to right, and y decreases top to bottom.

With WINDOW, you can redefine the screen border coordinates. You can also draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen with world coordinates. When you redefine the screen, you can draw graphics within a customized mapping system.

VBASICA converts world coordinates into physical coordinates for subsequent display within the current viewport. To make this transformation from world space to the physical space of the viewing surface (screen), you must know what portion of the (floating-point) world coordinate space contains the information to be displayed. This rectangular region in world coordinate space is a window.

RUN or WINDOW with no arguments disables window transformation. The WINDOW SCREEN variant inverts the normal Cartesian direction of the y coordinate.

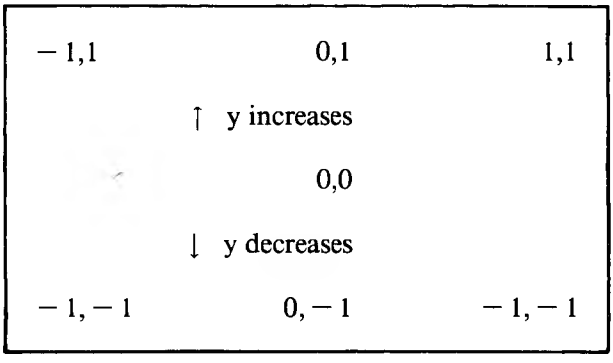
For example, in the default, a section of the screen appears as:



Now execute the following:

```
WINDOW (-1,-1)-(1,1)
```

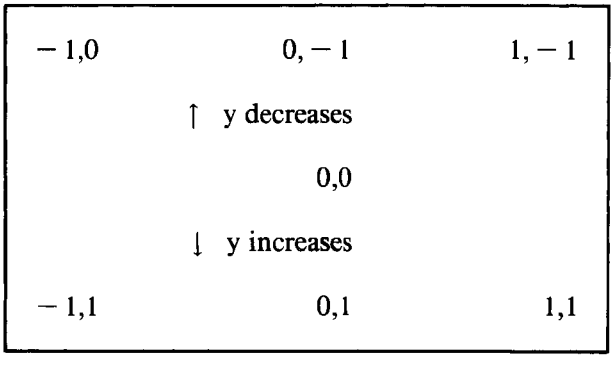
and the screen appears as:



If this variation is executed:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

then the screen appears as:



EXAMPLE:

The following example illustrates two lines with the same endpoint coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
200 LINE (100,100)-(150,150), 1
220 LOCATE 2,20:PRINT "The line on the default screen"
240 WINDOW SCREEN (100,100)-(200,200)
260 LINE (100,100)-(150,150), 1
280 LOCATE 8, 18:PRINT "amd the same line on a
    redefined window"
```


WRITE Statement

FORMAT:

WRITE [< expr list >]

PURPOSE:

Displays data on the screen.

REMARKS:

< expr list > is a list of numeric and/or string expressions. Commas separate the expressions. If < expr list > is omitted, a blank line is output. If < expr list > is included, the values of the expressions are displayed on your screen.

The output items are separated by commas. Strings are delimited by quotation marks. After the last item in the list appears, VBASICA inserts a carriage return/linefeed.

WRITE outputs numeric values using the same format as the PRINT statement.

EXAMPLE:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80,90,"THAT'S ALL"
Ok
```

WRITE# Statement

FORMAT:

WRITE# <filenum>,<expr list>

PURPOSE:

Writes data to a sequential file.

REMARKS:

<filenum> is the number under which a file was opened.

The expressions in <expr list> are string or numeric expressions, separated by commas.

Unlike PRINT#, WRITE# inserts commas between items as they are written to disk, and delimits strings with quotation marks. You do not need to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item is written to disk.

Assume that A$ is "CAMERA" and B$ is "93604-1". The statement:

```
WRITE#1,A$,B$
```

writes this to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

inputs "CAMERA" to A$ and "93604-1" to B$.

4

VBASICA and Communications

This chapter describes the VBASICA statements required to support asynchronous serial communication with other computers and peripherals, through the RS-232-C ports.

4.1 Communication I/O

Because you open the communications port as a file, all Input/output statements valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files: INPUT # < file number >, LINE INPUT # < file number >, and the INPUT$ function.

COM sequential output statements are the same as those for disk: PRINT # < file number > and PRINT # < file number > USING.

Refer to INPUT and PRINT for details of coding syntax and usage.

GET and PUT are only slightly different for COM files. See the GET and PUT statements for COM.

The TTY program that follows enables your VBASICA computer to be used as a conventional terminal. In addition to full-duplex communication with a host, the TTY program allows data to be downloaded to a file. Conversely, a file can be up-loaded (transmitted) to another machine.

In addition to demonstrating the elements of asynchronous communication, this program is useful in transferring VBASICA programs and data to and from your computer.

NOTE: The TTY program is set up to communicate using XON and XOFF. You may want to modify it for your environment.

4.2 The TTY Program

```
10 SCREEN 0,0:WIDTH 80
15 KEY OFF:CLS:CLOSE
20 DEFINT A-Z
25 LOCATE 25,1
30 PRINT STRING$(60," ")
40 FALSE=0:TRUE= NOT FALSE
50 MENU=5 ' Value of MENU key (CTRL-E)
60 XOFF$=CHR$(19):XON$=CHR$(17)

100 LOCATE 25,1:PRINT "Async TTY Program ";
110 LOCATE 1,1:LINE INPUT "Speed? ";SPEED$
120 COMFIL$="COM1:"+SPEED$+",E,7"
130 OPEN COMFIL$ AS #1

200 PAUSE=FALSE
210 A$=INKEY$: IF A$="" THEN 230
220 IF ASC(A$)=MENU THEN 300 ELSE PRINT #1,A$;
230 IF EOF(1) THEN 210
240 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
250 A$=INPUT$(LOC(1),#1)
260 PRINT A$;:IF LOC(1)>0 THEN 240
270 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
280 GOTO 210

300 LOCATE 1,1:PRINT STRING$(30," "):LOCATE 1,1
310 LINE INPUT"File? ";DSKFIL$

400 LOCATE 1,1:PRINT STRING$(30," "):LOCATE 1,1
410 LINE INPUT"(T)ransmit or (R)eceive? ";TXRX$
420 IF TXRX$="T" THEN OPEN DSKFIL$ FOR INPUT
    AS #2:GOTO 1000
430 OPEN DSKFIL$ FOR OUTPUT AS #2
440 PRINT #1,CHR$(13);
```

```
500 IF EOF(1) THEN GOSUB 600
510 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
520 A$=INPUT$(LOC(1),#1)
530 PRINT #2,A$;:IF LOC(1)>0 THEN 510
540 IF PAUSE THEN PAUSE=FALSE:PRINT #1,XON$;
550 GOTO 500

600 FOR I=1 TO 5000
610 IF NOT EOF(1) THEN I=9999
620 NEXT I
630 IF I>9999 THEN RETURN
640 CLOSE #2:CLS:LOCATE 25,10:PRINT
    "* Download complete *";
650 GOTO 200

1000 WHILE NOT EOF(2)
1010 A$=INPUT$(1,#2)
1020 PRINT #1,A$;
1030 WEND
1040 PRINT #1,CHR$(26); 'CTRL-Z to make close file.
1050 CLOSE #2:CLS:LOCATE 25,10:PRINT
     "** Upload complete **";
1060 GOTO 200

9999 CLOSE:KEY ON
```


4.3 Notes on the TTY Program

LINE NUMBER	COMMENTS
10	Sets the screen to Alpha mode and sets the width to 80.
15	Turns off the soft key display, clears the screen, and ensures all files are closed. **NOTE**: Asynchronous implies character I/O, as opposed to line or block I/O. Therefore, terminate all PRINTs, either to the COM file or to the screen, with a semicolon (;). This action suppresses the carriage return/line feed normally issued at the end of a PRINT statement.

LINE NUMBER	COMMENTS
20	Defines all numeric variables as INTEGER. Primarily for the subroutine at 600–620.
25–30	Clears the 25th line starting at column 1.
40	Defines Boolean TRUE and FALSE.
50	Defines the ASCII (ASC) value of the MENU key.
60	Defines the ASCII XON, XOFF characters.
100–130	Prints program ID and asks for baud rate (speed). Opens communications to file number 1, even parity, 7 data bits.
	Programmer exercise: Modify this section to check for valid baud rates.
200–280	Performs full-duplex I/O between the video screen and the device connected to the RS-232-C connector as follows:
	1. Read a character from the keyboard into A$. INKEY$ returns a null string if no character is waiting.
	2. If no character is waiting, then check if any characters are being received. If a character is waiting at the keyboard, and:
	— If the character is the MENU key, the user is ready to download a file, so get the filename.
	— If character (A$) is not the MENU key, send it by writing to the communication file (PRINT #1...).
	3. At 230 see if any characters are waiting in COM buffer. If not, then go back and check keyboard.
	4. At 240, if more than 128 characters are waiting, then set PAUSE flag saying we are suspending input. Send XOFF to host, stopping further transmission.
	5. At 250–260, read and display contents of COM buffer on screen until empty. Continue to monitor size of COM buffer (in 240). Suspend transmission if the program falls behind.
	6. Finally, resume host transmission by sending XON only if suspended by previous XOFF. Repeat process until MENU key is struck.
300–310	Get disk file name we are downloading to. Open file to tie number 2.
400–420	Asks if file named is to be transmitted (uploaded) or received (downloaded).

LINE NUMBER	COMMENTS
430	Sends a carriage return to the host to begin the download. This program assumes that the last command sent to the host was to begin such a transfer and was missing only the terminating carriage return. If a DEC system is the host, then such a command might be the following: **COPY TTY: = MANUAL.MEM < MENU key >** where the MENU key was struck instead of Return.
500	When no more characters are being received (LOC(x) returns 0), then perform a time-out routine (see line 600).
510	Again, if more than 128 characters are waiting, signal a pause and send XOFF to the host while up.
520–530	Read all characters in COM queue (LOC(x)) and write them to disk (PRINT #2..) until caught up.
540–550	If a pause was issued, restart host by sending XON and clear the pause flag. Continue process until no characters are received for a predetermined time.
600–650	This is the time-out subroutine. The FOR loop count was determined by experimentation. In short, if no character is received from the host for 17–20 seconds, then transmission is assumed complete. If any character is received during this time (line 610), then I is set well above FOR loop range to exit loop and then return to caller. If host transmission is complete, close the disk file and return to being a terminal.
1000–1060	Transmit routine. Until end of disk file do the following: Read one character into A$ with INPUT$ statement. Send character to COM device in 1020. Send a CTRL-Z at end of file in 1040 in case receiving device needs one to close its file. Finally, in lines 1050 and 1060, close our disk file, print completion message and go back to conversion mode in line 200.
9999	Presently not executed. As an exercise, add some lines to the routine 400–420 to exit the program via line 9999. This line closes the COM file left open and restores the soft key display.

4.4 The COM I/O Functions

The most difficult aspect of asynchronous communication is processing characters as fast as they are received. At rates above 2400 bps, character transmission must be suspended from the host long enough to catch up. This suspension can be done by sending XOFF (CTRL-S) to the host and XON (CTRL-Q) when ready to resume.

VBASICA provides three functions to help determine when an "overrun" condition is imminent:

LOC(x) Returns the number of characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the queue, LOC(x) returns 255. Because a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 255 characters remain in the queue, LOC(x) returns the actual count.

LOF(x) Returns the amount of free space in the input queue—that is, /C:<size> − LOC(x). You can use LOF to detect when the input queue is getting full. LOC is adequate for this purpose, as shown in the programming example.

EOF(x) If true (−1), indicates that the input queue is empty. Returns false (0) if any characters are waiting to be read.

These errors can occur:

1. "Communication Buffer Overflow" occurs if a read is attempted after the input queue is full (that is, LOC(x) returns 0).
2. "Device I/O Error" occurs if any of the following line conditions are detected on receive: Overrun Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.
3. "Device Fault" occurs if data set ready (DSR) is lost during I/O.

A

Error Messages

VBASICA provides the following error messages. The error codes and messages described in this appendix can appear when you are using the VBASICA Interpreter. Each message is explained and corrective measures are suggested, when appropriate.

CODE NUMBER	MESSAGE
1	**NEXT without FOR** A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
2	**Syntax error** A line contains an incorrect sequence of characters, such as unmatched parentheses, misspelled command or statement, incorrect punctuation, and so on.
3	**Return without GOSUB** A RETURN statement appears without a previous, unmatched GOSUB statement.
4	**Out of data** VBASICA executes a READ statement when no remaining DATA statements exist that contain unread data.
5	**Illegal function call** An out-of-range parameter is passed to a math or string function. An FC error can also occur as the result of: ▶ A negative or unreasonably large subscript ▶ A negative or zero argument with LOG ▶ A negative argument to SQR ▶ A negative mantissa with a noninteger exponent ▶ A call to a USR function for which the starting address was not given ▶ An incorrect argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO

CODE NUMBER	MESSAGE
6	**Overflow** The result of a calculation is too large to be represented in VBASICA number format. If underflow occurs, the result is zero and execution continues without an error.
7	**Out of memory** A program is too large, has too many FOR loops or GOSUBs, has too many variables, or contains expressions that are too complicated.
8	**Undefined line** A line reference in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE refers to a nonexistent line.
9	**Subscript out of range** An array element is referenced with a subscript outside the dimensions of the array or with the wrong number of subscripts.
10	**Redimensioned array** Two DIM statements are given for the same array, or a DIM statement is given for an array when the default dimension is 10.
11	**Division by zero** An expression contains a division by zero or the operation of involution raises zero to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution. In both cases, execution continues.
12	**Illegal direct** You have entered a statement that is illegal in direct mode.
13	**Type mismatch** A string variable name is assigned a numeric value (or vice versa), or a function that expects a numeric argument is given a string argument (or vice versa).
14	**Out of string space** String variables caused VBASICA to exceed the remaining free memory. VBASICA allocates string space dynamically until it runs out of memory.
15	**String too long** You have tried to create a string more than 255 characters long.
16	**String formula too complex** A string expression is too long or too complex. The expression must be broken into smaller expressions.

CODE NUMBER	MESSAGE
17	**Can't continue** You have tried to continue a program that: ▶ Has halted due to an error ▶ Modified during a break in execution ▶ Does not exist
18	**Undefined user function** A USR function is called before the function definition (DEF statement) is given.
19	**No RESUME** You entered an error-trapping routine that lacks a RESUME statement.
20	**RESUME without error** VBASICA encounters a RESUME statement before it enters an error-trapping routine.
21	**Unprintable error** An error message is not available for the error condition that exists. An error with an undefined error code usually causes this error.
22	**Missing operand** An expression contains an operator without a following operand.
23	**Line buffer overflow** You tried to input a line with too many characters.
24	**Device Timeout** VBASICA does not receive information from an I/O device within a predetermined amount of time.
25	**Device I/O Error** Fault status is returned from the parallel and serial devices. Usually indicates a hardware error in the printer or serial communications channel.
26	**FOR without NEXT** A FOR does not have a matching NEXT.
29	**WHILE without WEND** A WHILE statement does not have a matching WEND.
30	**WEND without WHILE** A WEND does not have a matching WHILE.
50	**Field overflow** A FIELD statement tried to allocate more bytes than were specified for the record length of a random file.

CODE NUMBER	MESSAGE
51	**Internal error** An internal malfunction has occurred in VBASICA. Report to your dealer the conditions under which the message appeared.
52	**Bad file number** A statement or command references a file with a file number that is not OPEN, or is out of the range of file numbers specified at initialization.
53	**File not found** A LOAD, KILL, or OPEN statement references a disk file that does not exist.
54	**Bad file mode** You have tried to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R.
55	**File already open** VBASICA issues a sequential output mode OPEN for a file already open, or a KILL is given for an open file.
57	**I/O error** An I/O error occurs on a device I/O operation. This error is fatal; the operating system cannot recover from the error. Formerly was Disk I/O error.
58	**File already exists** The filename specified in a NAME statement is identical to a filename already in use on the disk.
61	**Disk full** All disk storage space is in use.
62	**Input past end** VBASICA executes an INPUT statement for a null (empty) file, or after all the data in the file was INPUT. To avoid this error, use the EOF function to detect end-of-file.
63	**Bad record number** The record number in a PUT or GET statement is greater than the maximum allowed (32,767) or equal to zero.
64	**Bad file name** An illegal form is used for the filename in a LOAD, SAVE, KILL, or OPEN statement, for example, a filename with too many characters.
66	**Direct statement in file** VBASICA encountered a direct statement in the file while LOADing an ASCII-format file. VBASICA terminates the LOAD.

CODE NUMBER	MESSAGE
67	**Too many files** You tried to create a new file (using SAVE or OPEN) when all 255 directory entries are full.
68	**Device Unavailable** An attempt is made to open a file to a nonexistent device. Hardware may not exist to support the device, such as LPT2: or LPT3:, or you may have disabled it. This error occurs if VBASICA executes an OPEN "COM1:... statement but you disable RS-232-S with the /C:0 switch directive on the command line.
69	**Communication Buffer Overflow** VBASICA executes a communication input statement but the input queue is already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case several options are available: 1. Increase the size of the COM receive buffer with the /C: switch. 2. Implement a hand-shaking protocol with the host/satellite such as XON/XOFF, as demonstrated in the TTY programming example, to turn transmit off long enough to catch up. 3. Use a lower baud rate for transmit and receive.
70	**Disk Write Protect** One of the three hard disk errors returned from the diskette controller. This error occurs when you try to write to a write-protected diskette. Use an ON ERROR GOTO statement to detect this situation and request user action.
71	**Disk Not Ready** The diskette drive door is open or a diskette is not in the drive. Recover with an ON ERROR GOTO statement.
72	**Disk Media Error** Occurs when the FDC controller detects a hardware or media fault. This error usually indicates harmed media. Copy existing files to a new diskette and reformat the damaged diskette. FORMAT flags the bad tracks and places them in a file badtrack. The remainder of the diskette is now usable.

Extended Codes

Certain keys or combinations of keys cannot return a value within the ASCII code range. These keys are remapped to generate an extended code when VBASICA executes an INKEY$ statement. The codes returned by the INKEY$ statement consist of an ASCII null (00) as the first part of the two-byte string. If a two-byte string is received by INKEY$, then check the second key value to determine the key pressed. The ASCII code in decimal and the associated key(s) are:

SECOND CODE	MEANING
3	(null character) NUL
15	(shift tab) ←
16–25	ALT- Q, W, E, R, T, Y, U, I, O, P
30–38	ALT- A, S, D, F, G, H, J, K, L
44–50	ALT- Z, X, C, V, B, N, M
59–68	Function keys F1 through F10 (when not used as soft keys)
71	Home
72	Cursor Up
73	Pg Up
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	Pg Dn
82	Insert
83	Delete
84–93	F11–F20 (SHIFT- F1–F10)
94–103	F21–F30 (CTRL- F1–F10)
104–113	F31–F40 (ALT- F1–F10)
114	CTRL-PrtSc
115	CTRL-Cursor Left (previous word)
116	CTRL-Cursor Right (next word)
117	CTRL-End
118	CTRL-Pg Dn
119	CTRL-Home
120–131	ALT- 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	CTRL-Pg Up

ASCII Character Codes

ASCII Value		Screen	Code	
Decimal	Hex	Character	Name	Keystroke
000	00	(null)	NUL	—
001	01	☺	SOH	CTRL-A
002	02	☻	STX	CTRL-B
003	03	♥	ETX	CTRL-C
004	04	♦	EOT	CTRL-D
005	05	♣	ENQ	CTRL-E
006	06	♠	ACK	CTRL-F
007	07	(beep)	BEL	CTRL-G
008	08	◘	BS	CTRL-H
009	09	(tab)	HT	CTRL-I
010	0A	(line feed)	LF	CTRL-J
011	0B	(home)	VT	CTRL-K
012	0C	(form feed)	FF	CTRL-L
013	0D	(carriage return)	CR	CTRL-M
014	0E	♫	SO	CTRL-N
015	0F	☼	SI	CTRL-O
016	10	►	DLE	CTRL-P
017	11	◄	DC1	CTRL-Q
018	12	↕	DC2	CTRL-R
019	13	‼	DC3	CTRL-S
020	14	¶	DC4	CTRL-T
021	15	§	NAK	CTRL-U
022	16	▬	SYN	CTRL-V
023	17	↨	ETB	CTRL-W
024	18	↑	CAN	CTRL-X
025	19	↓	EM	CTRL-Y
026	1A	→	SUB	CTRL-Z
027	1B	←	ESC	Escape Key
028	1C	(cursor right)	FS	
029	1D	(cursor left)	GS	
030	1E	(cursor up)	RS	
031	1F	(cursor down)	US	

ASCII Value		Screen	ASCII Value		Screen
Decimal	Hex	Character	Decimal	Hex	Character
032	20	(space)	064	40	@
033	21	!	065	41	A
034	22	"	066	42	B
035	23	#	067	43	C
036	24	$	068	44	D
037	25	%	069	45	E
038	26	&	070	46	F
039	27	'	071	47	G
040	28	(	072	48	H
041	29	)	073	49	I
042	2A	*	074	4A	J
043	2B	+	075	4B	K
044	2C	,	076	4C	L
045	2D	-	077	4D	M
046	2E	.	078	4E	N
047	2F	/	079	4F	O
048	30	0	080	50	P
049	31	1	081	51	Q
050	32	2	082	52	R
051	33	3	083	53	S
052	34	4	084	54	T
053	35	5	085	55	U
054	36	6	086	56	V
055	37	7	087	57	W
056	38	8	088	58	X
057	39	9	089	59	Y
058	3A	:	090	5A	Z
059	3B	;	091	5B	[
060	3C	<	092	5C	\
061	3D	=	093	5D	]
062	3E	>	094	5E	^
063	3F	?	095	5F	_

ASCII Value		Screen	ASCII Value		Screen
Decimal	Hex	Character	Decimal	Hex	Character
096	60	`	128	80	Ç
097	61	a	129	81	ü
098	62	b	130	82	é
099	63	c	131	83	â
100	64	d	132	84	ä
101	65	e	133	85	à
102	66	f	134	86	å
103	67	g	135	87	ç
104	68	h	136	88	ê
105	69	i	137	89	ë
106	6A	j	138	8A	è
107	6B	k	139	8B	ï
108	6C	l	140	8C	î
109	6D	m	141	8D	ì
110	6E	n	142	8E	Ä
111	6F	o	143	8F	Å
112	70	p	144	90	É
113	71	q	145	91	æ
114	72	r	146	92	Æ
115	73	s	147	93	ô
116	74	t	148	94	ö
117	75	u	149	95	ò
118	76	v	150	96	û
119	77	w	151	97	ù
120	78	x	152	98	ÿ
121	79	y	153	99	Ö
122	7A	z	154	9A	Ü
123	7B	{	155	9B	¢
124	7C	|	156	9C	£
125	7D	}	157	9D	¥
126	7E	~	158	9E	Pt
127	7F	⌂	159	9F	ƒ

ASCII Value		Screen	ASCII Value		Screen
Decimal	Hex	Character	Decimal	Hex	Character
160	A0	á	192	C0	└
161	A1	í	193	C1	┴
162	A2	ó	194	C2	┬
163	A3	ú	195	C3	├
164	A4	ñ	196	C4	─
165	A5	Ñ	197	C5	┼
166	A6	ª	198	C6	╞
167	A7	º	199	C7	╟
168	A8	¿	200	C8	╚
169	A9	⌐	201	C9	╔
170	AA	¬	202	CA	╩
171	AB	½	203	CB	╦
172	AC	¼	204	CC	╠
173	AD	¡	205	CD	═
174	AE	«	206	CE	╬
175	AF	»	207	CF	╧
176	B0	▒	208	D0	╨
177	B1	▒	209	D1	╤
178	B2	▓	210	D2	╥
179	B3	│	211	D3	╙
180	B4	┤	212	D4	╘
181	B5	╡	213	D5	╒
182	B6	╢	214	D6	╓
183	B7	╖	215	D7	╫
184	B8	╕	216	D8	╪
185	B9	╣	217	D9	┘
186	BA	║	218	DA	┌
187	BB	╗	219	DB	■
188	BC	╝	220	DC	▄
189	BD	╜	221	DD	▌
190	BE	╛	222	DE	▐
191	BF	┐	223	DF	▀

C

ASCII Value		Screen
Decimal	Hex	Character
224	E0	α
225	E1	β
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	μ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	∅
238	EE	(
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	•
250	FA	·
251	FB	√
252	FC	n
253	FD	2
254	FE	■
255	FF	(blank 'FF')

Mathematical Functions

Functions not intrinsic to VBASICA are calculated as follows.

FUNCTION	VBASICA EQUIVALENT
Secant	SEC(X) = 1/COS(X)
Cosecant	CSC(X) = 1/SIN(X)
Cotangent	COT(X) = 1/TAN(X)
Inverse sine	ARCSIN(X) = ATN(X/SQR(− X*X + 1))
Inverse cosine	ARCCOS(X) = − ATN(X/SQR(− X*X + 1)) + 1.5708
Inverse secant	ARCSEC(X) = ATN(X/SQR(X*X − 1)) + SGN(SGN(X) − 1)*1.5708
Inverse cosecant	ARCCSC(X) = ATN(X/SQR(X*X − 1)) + (SGN(X) − 1)*1.5708
Inverse cotangent	ARCCOT(X) = ATN(X) + 1.5708
Hyperbolic sine	SINH(X) = (EXP(X) − EXP(− X))/2
Hyperbolic cosine	COSH(X) = (EXP(X) + EXP(− X))/2
Hyperbolic tangent	TANH(X) = EXP(− X)/EXP(X) + EXP(− X))*2 + 1
Hyperbolic secant	SECH(X) = 2/(EXP(X) + EXP(− X))
Hyperbolic cosecant	CSCH(X) = 2/(EXP(X) − EXP(− X))
Hyperbolic cotangent	COTH(X) = EXP(− X)/(EXP(X) − EXP(− X))*2 + 1
Inverse hyperbolic sine	ARCSINH(X) = LOG(X + SQR(X*X + 1))
Inverse hyperbolic cosine	ARCCOSH(X) = LOG(X + SQR(X*X − 1))
Inverse hyperbolic tangent	ARCTANH(X) = LOG((1 + X)/(1 − X))/2
Inverse hyperbolic secant	ARCSECH(X) = LOG((SQR(− X*X + 1) + 1)/X)
Inverse hyperbolic cosecant	ARCCSCH(X) = LOG((SGN(X)*SQR(X*X + 1) + 1)/X
Inverse hyperbolic cotangent	ARCCOTH(X) = LOG((X + 1)/(X − 1))/2

Converting Programs to VBASICA

If you have programs written in a BASIC other than VBASICA, some minor adjustments may be necessary before running them. This appendix describes specific things to look for when converting BASIC programs to VBASICA.

String Dimensions

Delete all statements that declare the length of strings. A statement such as DIM A$(I,J), which dimensions a one-dimensional string array for J elements of length I, should be converted to the VBASICA statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, the operator used in VBASICA for string concatenation.

In VBASICA, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms used by other BASICs, such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

OTHER BASIC	VBASICA
X$ = A$(I)	X$ = MID$(A$,I,1)
X$ = A$(I,J)	X$ = MID$(A$,I,J-I + 1)

If the substring reference is on the left side of an assignment and you use X$ to replace characters in A$, convert as follows:

OTHER BASIC	VBASICA
A$(I) = X$	MID$(A$,1,1) = X$
A$(I,J9) = X$	MID$(A$,I,J-I + 1) = X$

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. If this statement appears in a VBASICA program, VBASICA interprets the second equal sign as a logical operator, and sets B equal to −1 if C is equal to 0. To make sure that this statement is interpreted correctly, convert it to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With VBASICA, be sure all statements on a line are separated by a colon (:).

E

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

VBASICA Disk I/O

This appendix describes disk I/O procedures. If you are new to VBASICA or you are getting disk-related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements.

F.1 Program File Commands

Here is a review of the commands and statements used in program file manipulation.

```
SAVE <filename>[,A]
```

Writes the program in memory to a disk file. The A option writes the program as a series of ASCII characters; otherwise, VBASICA uses a compressed binary format.

```
LOAD <filename>[,R]
```

Loads a program from a disk file into memory. The R option runs the program immediately.

LOAD deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Programs can be chained or loaded in sections and still access the same data files.

LOAD < filename >,R and RUN < filename >,R are equivalent.

```
RUN <filename>[,R]
```

Loads the program from disk into memory and runs it. RUN deletes the current memory contents and closes all files before loading the program. If the R option is included, all open data files are kept open.

RUN < filename > ,R and LOAD < filename > ,R are equivalent.

```
MERGE <filename>
```

Loads the program from disk into memory without deleting the current memory contents. Line numbers used by the disk program are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the merged program resides in memory, and VBASICA returns to command level.

```
KILL <filename>
```

Deletes the specified file from the disk. < filename > can be a program file, or a sequential or random access data file.

```
NAME <old filename> AS <new filename>
```

Changes the name of a disk file. NAME can be used with program files, random files, or sequential files.

F.2 Protected File

To save a program in an encoded binary format, use the P (Protect) option with the SAVE command:


```
SAVE "MYPROG",P
```

A program saved in this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

F.3 Disk Data Files—Sequential and Random I/O

Two types of disk data files are created and accessed by a VBASICA program: sequential files and random access files.

F.3.1 Sequential Files

Although sequential files are easier to create than random files, they are not as fast or flexible when accessing the data. Data written to a sequential file is a series of ASCII characters stored one item after another (sequentially) in the order it is sent. Data is read back the same way.

The statements and functions used with sequential files are:

OPEN
CLOSE
PRINT#, PRINT# USING
EOF
INPUT#, LINE INPUT#
LOC
WRITE#

Follow these steps to recreate a sequential file and access the data in the file:

1. OPEN the file in O mode:

```
OPEN "O",#1,"DATA"
```

2. Write data to the file using the PRINT# statement. (WRITE# can be used instead.)

```
PRINT#1,A$;B$;C$
```

3. To access the data in the file, you must CLOSE the file and reopen it in "I" mode.

```
CLOSE #1
OPEN "I",#1,"DATA"
```

4. Use the INPUT# statement to read data from the sequential file into the program.

```
INPUT#1,X$,Y$,Z$
```

Here is a short program that creates a sequential file, "DATA", from information you enter at the keyboard:

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/83

NAME? MANFRED MANN
DEPARTMENT? KEYBOARD REPAIR
DATE HIRED? 08/16/83
```

The next program accesses the file created in the previous example and displays the name of everyone hired in 1983:

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="83" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
MANFRED MANN
Input past end in 20
Ok
```

The preceding program reads (sequentially) every item in the file. When all the data is read, line 20 causes an "Input past end" error. To avoid getting this error, insert the following line 15, which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

Then change line 40 to:

```
GOTO 15
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement:

```
PRINT#1, USING"####.##,";A,B,C,D
```

can be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

When used with a sequential file, the LOC function returns the number of sectors that have been written to or read from the file since it was OPENed. (A sector is a 128-byte block of data.)

Adding Data to a Sequential File

If you want to add data to the end of a sequential file residing on disk, you cannot simply open the file in O mode and start writing data. If you do this, you destroy the current contents of the sequential file. Instead, use the following procedure (used here to add data to an existing file called NAMES):

1. OPEN NAMES in Insert mode.
2. OPEN a second file called COPY in O mode.
3. Read in the data in NAMES and write it to COPY.
4. CLOSE NAMES and KILL it.
5. Write the new information to COPY.
6. Rename COPY as NAMES and CLOSE.
7. Now there is a disk file called NAMES that includes all previous data plus the new data you just added.

The next example illustrates this technique:

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$=""THEN 200 'CARRIAGE RETURN EXITS
    INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
```

```
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN
     "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

The preceding program can be used to create or add to a file called NAMES. This program also shows how to use LINE INPUT# to read strings that contain commas from the disk file. Remember, LINE INPUT# reads in characters from the disk until it sees a carriage return or it has read 255 characters. (It does not stop at quotation marks or commas.)

The error-trapping routine in line 2000 traps a "File does not exist" error in line 20. When this happens, the statements that copy the file are skipped, and COPY is created as a new file.

F.3.2 Random Files

You need more program steps to create and access random files than you do with sequential files; however, there are advantages to taking the extra trouble. One advantage is that random files require less room on the disk; VBASICA stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.) The biggest advantage to random files is that data is accessed randomly. You don't have to read through all the information, as you do with sequential files. Random access is possible because the information is stored and accessed in distinct units called records. Each record is numbered, making it easy for VBASICA to locate the record you need.

The statements and functions used with random files are:

OPEN
PUT
MKI$, MKS$, MKD$
FIELD
CLOSE
CVI, CVS, CVD
LSET/RSET
LOC
GET

Creating a Random File

Use these program steps to create a random file.

1. OPEN the file for random access (R mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables to be written to the random file.

```
FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be put into strings when put in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string; MKS$ for a single-precision value; and MKD$ for a double-precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Finally, use the PUT statement to write data from the buffer to the disk:

```
PUT #1,CODE%
```

The next example accepts input data from the keyboard and writes it to a random file:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Each time the PUT statement is executed, a record is written to the file. The two-digit code input in line 30 becomes the record number.

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Accessing a Random File

These program steps are required to access a random file:

1. OPEN the file in "R" mode:

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
```

   If a program does input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer:

```
GET #1,CODE%
```

4. The data in the buffer can now be accessed by the program. Convert numeric values back to numbers using the "convert" functions: CVI for integers; CVS for single-precision values; and CVD for double-precision values:

```
PRINT N$
PRINT CVS(A$)
```

Using this procedure, you can use a three-digit code to access and display records in the random file "FILE" created in the last example:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1,CODE%
50 PRINT N%
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

With random files, the LOC function returns the current record number. The current record number is computed by adding one to the number of the last record used in a GET or PUT statement. For example, this statement ends program execution if the current record number in file 1 is higher than 50:

```
IF LOC(1)>50 THEN END
```

The next program is an inventory program that uses random file access. In this program, the record number is also the part number, and it is assumed the inventory uses no more than 100 different part numbers. Lines 900–960 initialize the data file by writing CHR$(255) as the first character of each record. This character is used later (line 270 and line 500) to determine whether an entry already exists for that part number. Lines 130–220 show the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine. Here is the inventory program:

```
120 OPEN "R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
    "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)>255 THEN INPUT"OVERWRITE";A$:
    IF A$>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
```

```
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC (F$)-255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE$$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT"QUANTITY TO ADD";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT #1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;"IN STOCK":GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT"QUANTITY NOW";Q%;
    "REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)CVI(R$) THEN PRINT D$;"QUANTITY";
    CVI (Q$)TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100)THEN PRINT "BAD PART
    NUMBER":GOTO 840 ELSE GET #1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IFB$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

Index